Report No. NAWCADWAR-95005-4.5
Contract No. N62269-90-C-0569
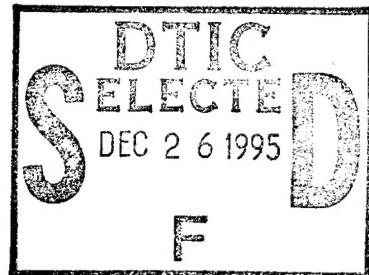
THE GAUSS MACHINE

Fred J. Taylor, Ph.D.
DEPARTMENT OF ELECTRICAL ENGINEERING
219 Grinter Hall
UNIVERSITY OF FLORIDA
Gainsville, FL 32611

1 NOVEMBER 1994

FINAL REPORT
Period Covering 25 September 1990 to 24 January 1992

19951219 011

NOTICES

REPORT NUMBERING SYSTEM - The numbering of technical project reports issued by the Naval Air Warfare Center, Aircraft Division, Warminster is arranged for specific identification purposes. Each number consists of the Center acronym, the calendar year in which the number was assigned, the sequence number of the report within the specific calendar year, and the official 2-digit correspondence code of the Functional Department responsible for the report. For example: Report No. NAWCADWAR-95010-4.6 indicates the tenth Center report for the year 1995 and prepared by the Crew Systems Engineering Department. The numerical codes are as follows.

Code	Department
4.1	Systems Engineering Department
4.2	Cost Analysis Department
4.3	Air Vehicle Department
4.4	Propulsion and Power Department
4.5	Avionics Department
4.6	Crew Systems Engineering Department
4.10	Conc. Analy., Eval. and Plan (CAEP) Department

PRODUCT ENDORSEMENT - The discussion or instructions concerning commercial products herein do not constitute an endorsement by the Government nor do they convey or imply the license or right to use such products.

Reviewed By: _Barry Kursch_ Date: _16 FEB 1995_
Author/COTR

Reviewed By: _____ Date: _2/16/95_
LEVEL III Manager

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of Information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204. Arlington, VA 22202-4302, and to the Office of management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 1 NOV 1994	3. REPORT TYPE AND DATES COVERED FINAL 9/25/90 – 1/24/92

4. TITLE AND SUBTITLE

THE GAUSS MACHINE

5. FUNDING NUMBERS

Contract No.
N62269-90-C-0569

6. AUTHOR(S)

FRED J. TAYLOR, PH.D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Department of Electrical Engineering
219 Grinter Hall
UNIVERSITY OF FLORIDA
Gainesville, FL 32611

8. PERFORMING ORGANIZATION REPORT NUMBER

UPN# 90081613

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Avionics Department, Engineering Division (Code 4.5.5.1)
NAVAL AIR WARFARE CENTER; AIRCRAFT DIVISION WARMINSTER
P.O. Box 5152
Warminster, PA 18974-0591

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

NAWCADWAR-95005-4.5

11. SUPPLEMENTARY NOTES

NAWCADWAR P.O.C. — BARRY J. KIRSCH — CODE 4.5.5.1

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Gauss machine is a SIMD systolic array architecture which takes advantage of the Galois-enhanced quadratic residue number system (GEQRNS) to form reduced complexity arithmetic elements. The Gauss machine is targeted at front-end signal and image processing applications. With a 2×2 array of GEQRNS multiplier-accumulators operating at 10 MHz the Gauss machine can achieve a peak equivalent throughput of 320 million operations per second when performing complex arithmetic. The Gauss machine is designed for a broader, more general class of problems than other RNS based systems which have been constructed: the Gauss machine may be used to accelerate computations which involve or may be expressed as matrix-matrix (level 3), matrix-vector (level 2), or vector-vector (level 1) operations. This paper describes the implementation of the Gauss machine and how it may be used to accelerate signal processing operations.

14. SUBJECT TERMS

SYSTOLIC ARRAY ARCHITECTURE, RESIDUE NUMBER SYSTEM, THE GAUSS MACHINE

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

THE GAUSS MACHINE: A GEQRNS DSP SYSTOLIC ARRAY

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

A REPORT PRESENTED TO THE NAVAL AIR WARFARE CENTER.
PREPARED BY THE HIGH-SPEED DIGITAL ARCHITECTURE
LABORATORY.

UNIVERSITY OF FLORIDA

1992

ABSTRACT

THE GAUSS MACHINE: A GEQRNS DSP SYSTOLIC ARRAY

The Gauss machine is a SIMD systolic array architecture which takes advantage of the Galois-enhanced quadratic residue number system (GEQRNS) to form reduced complexity arithmetic elements. The Gauss machine is targeted at front-end signal and image processing applications. With a 2×2 array of GEQRNS multiplier-accumulators operating at 10 MHz, the Gauss machine can achieve a peak equivalent throughput of 320 million operations per second when performing complex arithmetic. The Gauss machine is designed for a broader, more general class of problems than other RNS based systems which have been constructed: the Gauss machine may be used to accelerate computations which involve or may be expressed as matrix-matrix (level 3), matrix-vector (level 2), or vector-vector (level 1) operations. This paper describes the implementation of the Gauss machine and how it may be used to accelerate signal processing operations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Theory

Chapter 1

BASIS OF RESIDUE NUMBER SYSTEM

1.1 The Chinese Remainder Theorem

There are two large penalties in performing arithmetic in the two's complement system: the carry must propagate across the entire word for addition operations, and the size of the multiplier grows as the square of the width of the word. The Chinese Remainder Theorem (CRT) [1, 2] suggests a means of eliminating the carry propagation problem and of producing a multiplier that grows linearly with the width of the word. The CRT is presented below.

Theorem 1 (The Chinese Remainder Theorem) *Let $M = \prod_{i=1}^{L} p_i$, where for $i, j \in \{1, 2, 3, \ldots, L\}$, $\gcd(p_i, p_j) = 1$ for all $i \neq j$, and each $p_i \in \mathbf{Z}^+$. Then there exists an isomorphism $\phi\colon \mathbf{Z}_M \leftrightarrow \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L}$ described by the following.*

Let $m_i = M/p_i$, and $m_i m_i^{-1} \equiv 1 \pmod{p_i}$ for all $i \in \{1, 2, 3, \ldots, L\}$. If $X \in \mathbf{Z}_M$, let $\phi(X) = (x_1, x_2, x_3, \ldots, x_L)$ where $x_i \equiv X \pmod{p_i}$ for all $i \in \{1, 2, 3, \ldots, L\}$ then $X = \phi^{-1}(x_1, x_2, x_3, \ldots, x_L)$ is described by the following congruence

$$X \equiv \left\{ \sum_{i=1}^{L} m_i < m_i^{-1} x_i >_{p_i} \right\} \pmod{M}$$

where $< \bullet >_p$ indicates the unary $\pmod{p}$ operation.

The CRT forms the basis for the RNS. In the RNS, two's complement integers are converted to their L-tuple residue representation by the ring isomorphism ϕ:

$\mathbf{Z}_M \leftrightarrow \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L}$ described by the CRT. The numbers which are in their L-tuple representation may be added and multiplied component-wise and reconstructed via the CRT to form the correct result in $\mathbf{Z}_M$. For example, consider the RNS system described by $p_1 = 3$, $p_2 = 5$, and $p_3 = 7$. Then $M = p_1 p_2 p_3 = 105$. Let $a = 7$, and $b = 9$ where $a, b \in \mathbf{Z}_M$. The numbers a and b may be mapped to their RNS 3–tuple representation via the mapping ϕ:

$$\phi(a) = (<7>_3, <7>_5, <7>_7) = (1, 2, 0)$$
$$\phi(b) = (<9>_3, <9>_5, <9>_7) = (0, 4, 2).$$

Arithmetic may be performed on the RNS L–tuple representation of $a, b \in \mathbf{Z}_M$ given by the mapping ϕ. Let $\phi(a) = (a_1, a_2, a_3, \ldots, a_L)$, and $\phi(b) = (b_1, b_2, b_3, \ldots, b_L)$. Then

$$\phi(a \circ b) = (<a_1 \circ b_1>_{p_1}, <a_2 \circ b_2>_{p_2}, <a_3 \circ b_3>_{p_3}, \ldots, <a_L \circ b_L>_{p_L}),$$

where $\circ \in \{+, -, \times\}$. Consider the 3–tuple representations of a and b:

$$(1, 2, 0) + (0, 4, 2) = (<1+0>_3, <2+4>_5, <0+2>_7) = (1, 1, 2) \qquad (1.1)$$
$$(1, 2, 0) \times (0, 4, 2) = (<1 \cdot 0>_3, <2 \cdot 4>_5, <0 \cdot 2>_7) = (0, 3, 0). \qquad (1.2)$$

For comparison, the mapping of $a + b = 16$ and $ab = 63$ to their RNS 3–tuple representation:

$$\phi(a + b) = (<16>_3, <16>_5, <16>_7) = (1, 1, 2) \qquad (1.3)$$
$$\phi(ab) = (<63>_3, <63>_5, <63>_7) = (0, 3, 0) \qquad (1.4)$$

The operations performed on the RNS representations of a and b (equations 1.1,1.2) give the same results as the RNS representation of $a + b$ and ab (equations

1.3, 1.4) performed in $\mathbf{Z}_M$. Now consider the restoration of the representation of $a + b, ab \in \mathbf{Z}_M$ from the RNS representations. For $(p_1, p_2, p_3) = (3, 5, 7)$ we have $m_1 = 35, m_1^{-1} = 2, m_2 = 21, m_2^{-1} = 1, m_3 = 15$, and $m_3^{-1} = 1$. From above we have $\phi(a + b) = (1, 1, 2)$, and $\phi(ab) = (0, 3, 0)$.

$$
\begin{aligned}
\phi^{-1}(1,1,2) &= \left\{ \sum_{i=1}^{3} m_i < m_i^{-1} x_i >_{p_i} \right\} \quad (\text{mod } 105) \\
&= \{35 < 2 \cdot 1 >_3 + 21 < 1 \cdot 1 >_5 + 15 < 1 \cdot 2 >_7\} \quad (\text{mod } 105) = 16 \\
\phi^{-1}(0,3,0) &= \left\{ \sum_{i=1}^{3} m_i < m_i^{-1} x_i >_{p_i} \right\} \quad (\text{mod } 105) \\
&= \{35 < 2 \cdot 0 >_3 + 21 < 1 \cdot 3 >_5 + 15 < 1 \cdot 0 >_7\} \quad (\text{mod } 105) = 63
\end{aligned}
$$

Thus we see that the results produced by the mapping ϕ^{-1} are as expected. Generally, the moduli are chosen to be small enough that the adders and multipliers may be implemented in a reasonably small memory-based lookup table. In a VLSI implementation we might leverage advanced memory technology and thereby achieve greater speed and smaller die area.

1.2 Complex Residue Number System (CRNS)

The RNS may be used to perform computations with complex numbers by using RNS arithmetic elements to emulate the operations which would be performed using two's complement hardware. The use of RNS arithmetic to perform complex operations is called complex RNS or CRNS. Suppose we have Gaussian integers $a + jb, c + jd \in \mathbf{Z}_M[j]/(j^2 + 1)$, and ψ denotes the isomorphism between the Gaussian integers and the CRNS: $\psi: \mathbf{Z}_M[j]/(j^2 + 1) \leftrightarrow \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L} \times \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L}$. Then

$$
(a + jb) + (c + jd) = (a + c) + j(b + d)
$$

$$= \psi^{-1}\{\psi(a) + \psi(b)\} + j\psi^{-1}\{\psi(b) + \psi(d)\}$$

$$(a + jb) \times (c + jd) = (ac - bd) + j(ad + bc)$$

$$= \psi^{-1}\{\psi(a)\psi(c) - \psi(b)\psi(d)\} + j\psi^{-1}\{\psi(a)\psi(d) + \psi(b)\psi(c)\}.$$

While the complex addition takes only two additions, the complex multiplication takes four multiplications and two additions: the CRNS requires the same number of additions and multiplications as the Gaussian integers.

1.3 Quadratic Residue Number System (QRNS)

The QRNS [3, 4] is a variation upon the RNS which allows complex additions to be performed with two RNS additions and complex multiplications to be performed with two RNS multiplications. This enhancement is accomplished by encoding the real and imaginary components into two independent components. Given a prime p of the form $p = 4k + 1$ where $k \in \mathbf{Z}$ then the congruence $x^2 \equiv -1 \pmod{p}$ has two solutions in the ring $\mathbf{Z}_p$ that are multiplicative and additive inverses of one another. Let $\hat{j}$ and $\hat{j}^{-1}$ denote the two solutions to the above congruence. Define a mapping $\theta\colon \mathbf{Z}_p[j]/(j^2 + 1) \to \mathbf{Z}_p \times \mathbf{Z}_p$ by

$$\theta(a + jb) = (z, z^*)$$

$$z \equiv (a + \hat{j}b) \pmod{p}$$

$$z^* \equiv (a - \hat{j}b) \pmod{p}.$$

Furthermore, the inverse mapping $\theta^{-1}\colon \mathbf{Z}_p \times \mathbf{Z}_p \to \mathbf{Z}_p[j]/(j^2 + 1)$ is given by

$$\theta^{-1}(z, z^*) = <2^{-1}(z + z^*)>_p + j<2^{-1}\hat{j}^{-1}(z - z^*)>_p .$$

Suppose $(z, z^*), (w, w^*) \in \mathbf{Z}_p \times \mathbf{Z}_p$. Then the addition and multiplication

operations in the ring $< \mathbf{Z}_p \times \mathbf{Z}_p, +, \cdot >$ are given by

$$(z, z^*) + (w, w^*) = (z + w, z^* + w^*)$$

$$(z, z^*)(w, w^*) = (zw, z^* w^*).$$

For example, consider a QRNS system with moduli $p_1 = 5$ and $p_2 = 13$. Let the Gaussian integers $u, v \in \mathbf{Z}[j]/(j^2+1)$ be given as $u = 5+j3$, and $v = 4+j3$. In $\mathbf{Z}_5$ we have $\hat{j}_1 = 2$ and $\hat{j}_1^{-1} = 3$. It can be seen that 2 and 3 are additive and multiplicative inverses of each other in $\mathbf{Z}_5$ and also satisfy the congruence $x^2 \equiv -1 \pmod{5}$. In $\mathbf{Z}_{13}$ we have $\hat{j}_2 = 5$ and $\hat{j}_2^{-1} = 8$. Also, $2^{-1} \equiv 3 \pmod{5}$, and $2^{-1} \equiv 7 \pmod{13}$. Therefore the QRNS representations of u and v are given by

$$\theta(u) = (z_u, z_u^*)$$

$$z_u = (< 5 + \hat{j}_1 3 >_5, < 5 + \hat{j}_2 3 >_{13}) = (1, 7)$$

$$z_u^* = (< 5 - \hat{j}_1 3 >_5, < 5 - \hat{j}_2 3 >_{13}) = (4, 3)$$

$$\theta(v) = (z_v, z_v^*)$$

$$z_v = (< 4 + \hat{j}_1 3 >_5, < 4 + \hat{j}_2 3 >_{13}) = (0, 6)$$

$$z_v^* = (< 4 - \hat{j}_1 3 >_5, < 4 - \hat{j}_2 3 >_{13}) = (3, 2).$$

The arithmetic operations in the QRNS are performed in the same manner as in the RNS. For example:

$$\theta(u) + \theta(v) = (z_u + z_v, z_u^* + z_v^*) = (z_{u+v}, z_{u+v}^*)$$

$$z_{u+v} = (< 1 + 0 >_5, < 7 + 6 >_{13}) = (1, 0)$$

$$z_{u+v}^* = (< 4 + 3 >_5, < 3 + 2 >_{13}) = (2, 5)$$

$$\theta(u)\theta(v) = (z_u z_v, z_u^* z_v^*) = (z_{uv}, z_{uv}^*)$$

$$z_{uv} = (< 1 \cdot 0 >_5, < 7 \cdot 6 >_{13}) = (0, 3)$$

$$z_{uv}^* = (<4 \cdot 3>_5, <3 \cdot 2>_{13}) = (2,6).$$

For comparison, note that $uv = 11 + j27$ and $u + v = 9 + j6$. The QRNS representations of uv and $u + v$ are given as

$$\theta(u + v) = (z_{u+v}', z_{u+v}^{*\prime})$$

$$z_{u+v}' = (<9 + \hat{j}_1 6>_5, <9 + \hat{j}_2 6>_{13}) = (1,0)$$

$$z_{u+v}^{*\prime} = (<9 - \hat{j}_1 6>_5, <9 - \hat{j}_2 6>_{13}) = (2,5)$$

$$\theta(uv) = (z_{uv}', z_{uv}^{*\prime})$$

$$z_{uv}' = (<11 + \hat{j}_1 27>_5, <11 + \hat{j}_2 27>_{13}) = (0,3)$$

$$z_{uv}^{*\prime} = (<11 - \hat{j}_1 27>_5, <11 - \hat{j}_2 27>_{13}) = (2,6).$$

The above results for the QRNS representations $\theta(uv)$ and $\theta(u + v)$ agree with $\theta(u)\theta(v)$ and $\theta(u) + \theta(v)$ computed in the QRNS representation. The isomorphism θ is generally implemented by a combination of arithmetic elements and table lookup. Since the z and z^* channels are independent we are able to easily construct parallel hardware to perform operations on both channels at the same time without any communication between the channels. This parallelism allows us to easily perform a complex addition or multiplication in one cycle. While parallel hardware would allow us to perform a CRNS addition in one cycle, the multiplication in the CRNS requires two additions and four multiplications. Using the same amount of hardware as a QRNS multiplier-accumulator, a CRNS multiplier-accumulator would take twice as many cycles to complete a single multiply-accumulate operation.

1.4 Galois Enhanced QRNS (GEQRNS)

The QRNS requires us to implement a multiplier which takes N bit inputs and produces an N bit output. The multiplier could be implemented using either a direct implementation with modular correction or a lookup table. The primary disadvantage of this is that despite the small size of the RNS adder, the multiplier is still large. We may take advantage of the properties of Galois fields [5] to simplify the implementation of an RNS multiplier.

For any prime modulus p there exists some $\alpha \in \mathbf{Z}_p$ that generates all non-zero elements of the field $GF(p)$. That is to say $\{\alpha^i \mid i = 0, 1, 2, \ldots, p - 2\} = GF(p) \setminus 0$. Thus, we may uniquely represent all non-zero elements of $\mathbf{Z}_p$ by their exponents. These number theoretic logarithms may be added modulo $p-1$ to produce multiplication: $\alpha^{<i+j>_{p-1}} = < \alpha^i \alpha^j >_p$. Note that since zero is not an element of $GF(p) \setminus 0$ the zero must be handled as an exception. Practically, this means that the inputs must be checked before the number theoretic logarithm to determine whether either one is a zero, and if one of the inputs is a zero, then the output of the multiplier should be set to zero.

For example, suppose that $p = 7$. Then $\alpha = 3$ generates $GF(7) \setminus 0$: $\{3^i \mid i = 0, 1, 2, 3, 4, 5\} = \{1, 3, 2, 6, 4, 5\}$. Suppose we wish to multiply 2 and 3. First we would take the number theoretic logarithm of 2 and 3 to the base $\alpha = 3$:

$$\log_3(2) = 2 \iff 3^2 \equiv 2 \pmod 7$$
$$\log_3(3) = 1 \iff 3^1 \equiv 3 \pmod 7.$$

In order to multiply 2 and 3 we now add the number theoretic logarithms modulo $p - 1$:

$$2 \cdot 3 = < 3^2 \cdot 3^1 >_7 = < 3^{<2+1>_6} >_7 = < 3^3 >_7 = 6.$$

The architecture of a GEQRNS multiplier is illustrated in Figure 1.1 without the zero detection and handling indicated. The multiplier requires two duplicate N-entry memories to perform the number theoretic logarithm, and an $N + 1$-entry table to perform the modulo $p - 1$ correction and number theoretic exponentiation. Note that while the modulo $p - 1$ correction and number theoretic exponentiation represent two separate steps, they may be integrated into a single table. Typically, the multiplicands will be converted to the GEQRNS number theoretic logarithm form by the conversion engine which computes the residues of the integer inputs.
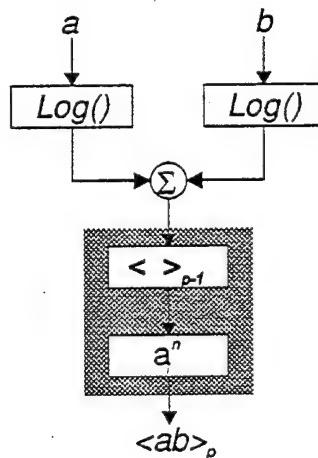
Figure 1.1: Block Diagram of a GEQRNS Multiplier

1.5 L-CRT

The L-CRT [1, 2] offers an alternative to the CRT which has the advantage of integrating scaling into the CRT and avoiding the need for a modulo M adder. The L-CRT is computed by factoring M into a real scale factor V and an integer $M' = 2^k$, where $k \in \mathbf{Z}^+$, such that $M = VM'$, and $0 < M' < M$. Additionally, as for the

CRT, $m_i = M/p_i$. The L-CRT is given as

$$X_S = \left\{ \sum_{i=1}^{L} \lfloor m_i < m_i^{-1} x_i >_{p_i} /V \rfloor \right\} \quad (\text{mod } M'),$$

where $\lfloor \bullet \rfloor$ denotes the least integer or floor function. Since $M' = 2^k$ where $k \in \mathbf{Z}^+$ we may compute the sum X_S using regular k–bit two's complement adders. The $\lfloor m_i < m_i^{-1} x_i >_{p_i} /V \rfloor$ term for any fixed set of moduli is dependent only upon x_i and thus may be generated using a small, fast memory based table lookup. The disadvantage of the L-CRT is that it may introduce an error into the computed X_S. The error in the L-CRT is given by $0 \leq |X/V - X_S| < L$. For front-end signal processing applications this error is not critical since $L \ll M$. A block diagram of two L-CRT engines is shown in Figure 1.2.

The L-CRT has the advantage of avoiding the modulo M adder required to implement the 'true' CRT and provides a means of scaling without additional hardware. For VLSI and discrete implementations this advantage is particularly important since division, like multiplication, are space-time intensive and cannot be performed in the RNS since it is division-free.
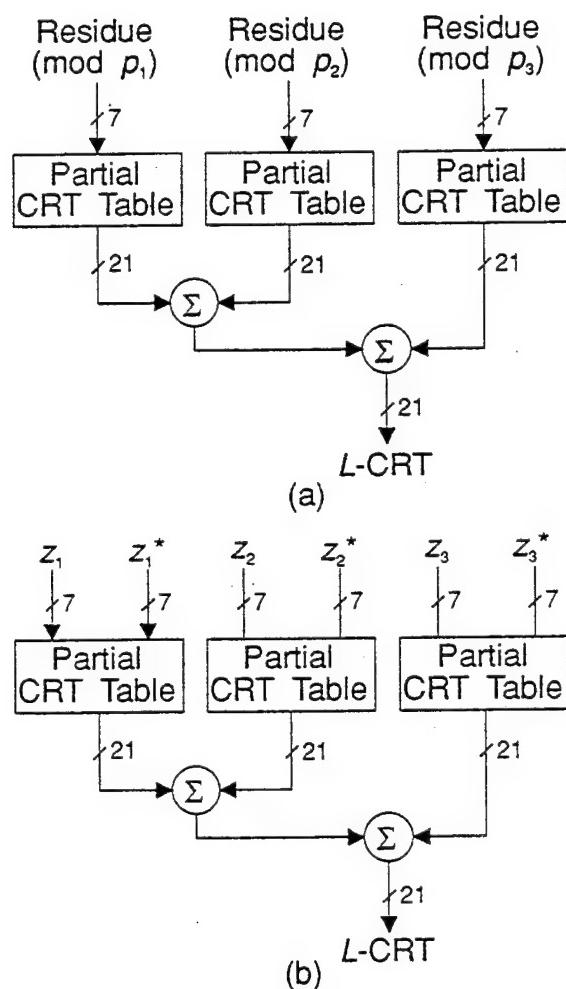
Figure 1.2: Block Diagram of L-CRT (a) and QRNS Augmented L CRT (b)

Part II

InvestiGATOR Array Processor Backplane

Chapter 2

INTRODUCTION

2.1   Motivation

There exists a need for an environment appropriate to the task of developing ex-
perimental array processors. This need is indicated by the large I/O requirements
and physical size of experimental array processors. Traditional environments such as
personal computers or larger systems such as the VME bus are not appropriate as
they lack adequate space and I/O capabilities. Thus the motivation is established for
the development of a testbed for experimental array processors.

Additional capabilities are desirable. In particular, beyond the need to solve
physical form factor problems and I/O bandwidth bottlenecks, there is an additional
desire that the system should be host independent. The ideal host interface for
achieving host independence is the SCSI interface. The SCSI interface exists on
all common personal computers and workstations. There also exist a number of
peripherals which may take advantage of the SCSI interface, thus allowing the testbed
to utilize a number of mass storage and data acquisition products.

2.2   Design Parameters

Given the motivation presented in the previous section, the design parameters are
described as follows. The control of the array processor and the SCSI interface require
substantial machine intelligence. Thus the selection of a microprocessor is required.

The Motorola 68030 was selected since it is capable of sustaining block data moves of approximately forty megabytes per second (at 20 MHz), and because of previous design experience with the 68000 family. First generation SCSI controller chips such as the NCR 8350 require substantial processor intervention in order to operate: each byte transferred causes an interrupt to occur. Additionally, these first generation SCSI controller chips were only capable of asynchronous operation at data rates of approximately 1.6 megabytes per second while many hosts operate synchronously at a maximum data rate of five megabytes per second. A second generation device was selected, the Western Digital 33C93A. The WD33C93A (second sourced by Advanced Micro Devices and sometimes referred to as the Am33C93A) executes SCSI commands independently of the host processor and is capable of transmitting large quantities of data without host intervention. For purposes of debugging, the array processor testbed also features RS-232C serial communications.

Memory requirements for the testbed are modest. The testbed need only buffer data transactions between the host and array and perform some translation of commands from the host to the array. Thus it was determined that the testbed processor would only require one megabyte of high speed RAM and 128 kilobytes of ROM. Since the processor typically is moving large, contiguous blocks of data between the SCSI processor and the array processor, a memory architecture which performs well in block operations is desirable. A dynamic RAM variant called static-column RAM (SCRAM) is particularly well suited to this task. The SCRAM is fundamentally a standard DRAM, however, once the row address has been latched into the device, the device operates as a static RAM for all subsequent accesses to that row of memory. These accesses may occur until refresh is required. The advantage to this means of memory operation are that a 70 ns device offers 35 ns

access times during static column operation. The static column mode of operation is synergistic with the 68030's burst mode of operation. Using the burst mode of operation the 68030 may read four longwords with reduced penalty. In particular, in the burst mode of operation, the worst-case first word read time is two clock cycles (at 20 MHz, $T_{cycle} = 50$ns). Subsequent accesses in non-burst mode still execute in two clock cycles. Subsequent burst-mode accesses execute in one clock cycle. Thus, the maximum memory bandwidth without burst access is forty megabytes per second, while the maximum memory bandwidth with burst access is sixty-four megabytes per second.

Chapter 3

IMPLEMENTATION

This chapter describes the implementation of the InvestiGATOR array pro-
cessor testbed. The description is broken into modules reflecting the various major
components of the backplane: the CPU, the memories, the I/O components, the
array interface, and remaining miscellaneous material.

3.1   Architecture

The InvestiGATOR backplane and SCSI control processor is constructed from sev-
eral discrete blocks. These blocks may be divided into four groups. The first, the
CPU is based upon the Motorola MC68030. The second, the memory, consists of
one megabyte of high performance static-column RAM, and 128 kilobytes of low
performance EPROM. The third group is the I/O module which includes a high per-
formance SCSI port, dual RS-232C serial ports, and an I/O expansion port. The
fourth group is the array bus and interface. A block diagram of the InvestiGATOR
is shown in Figure 3.1.

The SCSI port is a single-ended, eight-bit implementation supporting syn-
chronous transfers up to five megabytes per second. The SCSI port has a local
thirty-two kilobyte buffer which allows the central processor to operate without in-
terference while transfers are underway. SCSI packets may be transferred either to or
from the InvestiGATOR with as few as two interrupts of the central processor. This
autonomous operation allows the CPU to dedicate a large percentage of its processing
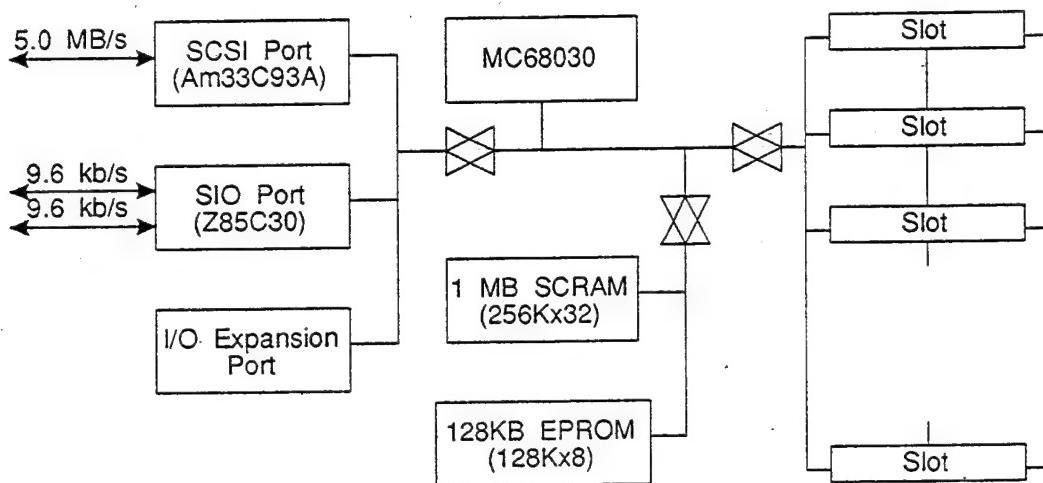
Figure 3.1: Block Diagram of the InvestiGATOR Array Processor Testbed

budget to servicing the attached experimental array processor.

The serial port supports two RS-232C channels with programmable baud rates of up to 9600 bps. The serial port is intended to act primarily as a debugging tool. The I/O expansion port has a full thirty-two bit data bus, twenty-bit address bus, and interrupt capabilities. This bus may be used to attach data acquisition, additional I/O capabilities, or memory.

The RAM block is based upon static-column RAM supporting synchronous and burst-mode accesses. This memory offers very high performance in block transfers.

3.2 CPU Module

This section describes the generation of the various signals which are used in the CPU module to service the MC68030, and signals which are used to interface with external devices and busses. This section refers to schematics which are found in Appendix A. A block diagram of the CPU module with its major subsystems is

shown in Figure 3.2.
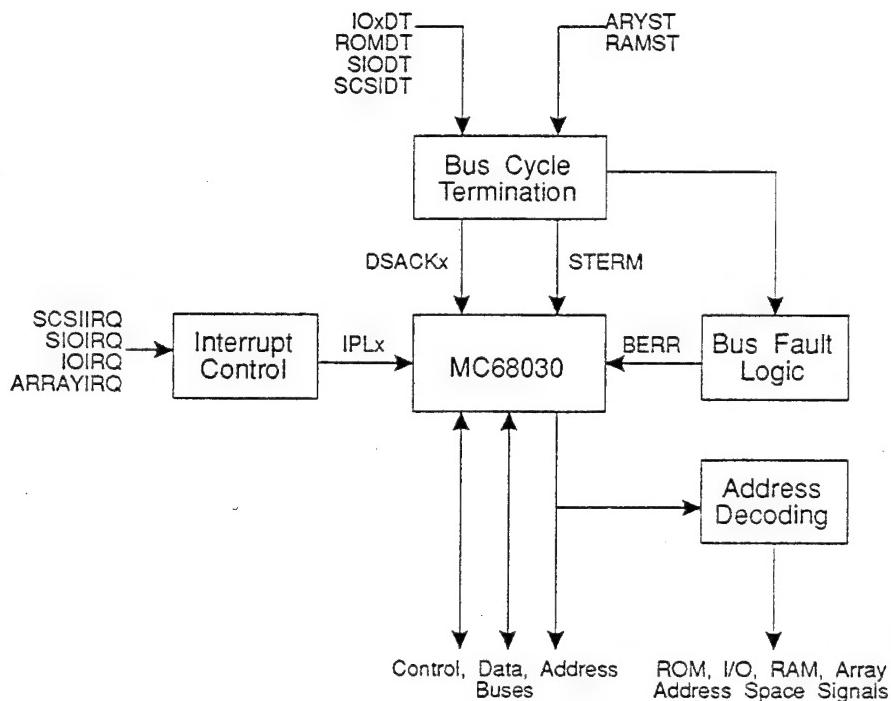
Figure 3.2: Block Diagram of CPU Module

3.2.1   Cache Control

The MC68030 provides a mechanism whereby external circuitry may indicate to the 68030 which addresses are cachable, the cache inhibit input, CIIN*. CIIN* is generated by PAL0 and inhibits the cache when accessing the I/O and array addressing spaces. Additionally, the 68030 provides a means for disabling the cache from external hardware, primarily for debugging purposes. This is the cache disable input, CDIS*. CDIS* may be asserted or negated using switch S3.

The primary reason for the selection of the MC68030 as the control processor of the InvestiGATOR was its on-chip instruction/data cache and burst cache fill mechanism. The 68030 provides a means of bursting four longwords of instruc-

tions or data into the cache. This is accomplished using the MC68030's cache burst request/acknowledge (CBREQ*/CBACK*) handshaking protocol. When the 68030 runs a bus cycle in which it can execute a burst fill of the cache it asserts the CBREQ* signal. If the addressed device wishes to proceed with a burst fill of the cache it must acknowledge the burst request with CBACK*. In a zero wait state system the 68030 can read four longwords in eight cycles (*i.e.*, forty megabytes per second) using standard bus cycles while the same four longwords can be read in five clock cycles (*i.e.*, sixty-four megabytes per second) using burst mode. There is support for burst filling of the cache from the RAM module only (see Table 3.2). A burst acknowledge on the part of the RAM module is passed through a D flip-flop clocked 180 degrees out of phase with the 20 MHz system clock in order to stretch the CBACK* signal.

3.2.2 Interrupt Control

The MC68030 provides a seven level prioritized interrupt mechanism using the IPL0-2* signals. PAL1 provides priority encoding of the various interrupt signals generated in the InvestiGATOR. The majority of the signals are provided to I/O devices, however, there is also an interrupt line reserved for the array bus. The prioritization of the interrupt sources is given below:

Request Priority	Description
7	NMI (Non-Maskable Interrupt). Reserved.
6	SCSI Port.
5	Reserved.
4	SIO port.
3	Reserved.
2	I/O Bus.
1	Array Bus.

Table 3.1: Interrupt Priority Levels

The InvestiGATOR uses the MC68030's interrupt autovector mechanism to vector interrupts. This is accomplished by asserting the AVEC* input of the 68030 when an interrupt acknowledge cycle is executed. AVEC* is generated by PAL0 using a clocked output. AVEC* is asserted when PAL0 detects an interrupt acknowledge cycle. The 68030 also provides one additional signal related to interrupts, the IPEND* (interrupt pending signal). The IPEND* signal is not used by the InvestiGATOR.

3.2.3 Address Space Decoding

Address space decoding is provided by PAL0. PAL0 decodes four primary address spaces: RAM space, ROM space, I/O space, and array space. These address space signals are address strobe qualified. This address space arrangement consumes sixty-four megabytes of the four gigabyte available address space, however, the sixty-four megabyte space is repeated (*i.e.*, A26-A31 are ignored). Accesses to memory spaces besides program and data space are ignored (with the exception of interrupt acknowledge cycles which run in CPU space) and will result in a bus fault after a timeout. Address space decoding is summarized in Table 3.2.

	Address Range	Description
C	0h—1FFFFh	ROM space.
	20000h—FFFFFh	I/O space.
B,C	100000h—FFFFFFh	RAM space.
	1000000h—3FFFFFFh	Array space.

C=cachable, B=burst cycle support.

Table 3.2: Address Space Decoding

3.2.4 Bus Cycle Termination

The 68030 provides two mechanisms for normal termination of bus cycles: asynchronous termination and synchronous termination. Both means of termination are supported by the InvestiGATOR. The synchronous termination mechanism is a high speed termination mechanism for use with thirty-two bit data ports only. In practice, only the RAM space and array space use synchronous termination. The MC68030's synchronous termination input, STERM* is generated by taking the logical OR of the two possible sources of synchronous termination requests, and then using a D flip-flop clocked 180 degrees out of phase with the 20 MHz system clock to stretch the STERM* signal, see Figure 3.3
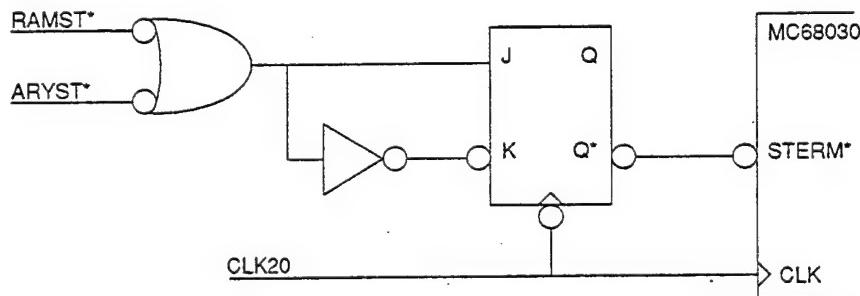
Figure 3.3: STERM Signal Input and Conditioning

The asynchronous bus cycle termination mechanism allows for dynamic bus sizing for eight, sixteen, and thirty-two bit ports. The asynchronous termination signals, DSACK0* and DSACK1*, are provided by PAL1A which generates the appropriate DSACKs for various ports (primarily I/O).

3.2.5 Abnormal Bus Cycle Termination: Bus Error Control

It is possible to attempt to access addresses for which there is no corresponding device. In this event it is necessary for external circuitry to terminate the bus cycle.

Additionally, it may be desirable to terminate an I/O or array bus cycle with an error condition. Bus cycles may be terminated with a fault condition by assertion of the MC68030's BERR* signal. Assertion of the BERR* signal is controlled by the BERR control state machine, located on MACH2. This state machine tracks bus cycles and asserts BERR* when the I/O or array busses request, or in the event of a timeout, indicated by the trickle count output of an eight bit watchdog timer (counter). A state machine diagram is given in Figure B.1.

3.2.6   Byte Select Signals

The CPU module provides byte select signals (UU*, UM*, LM*, and LL*) to external modules by decoding the A0, A1, SIZ0, and SIZ1 outputs of the 68030. These byte selects are decoded by PAL1A and are not qualified by the address strobe.

3.2.7   Miscellaneous Signals

The InvestiGATOR does not support multiple bus mastering in the controller so the BR* (bus request) input is negated. The BG* (bus grant) signal is ignored and the BGACK* (bus grant acknowledge) signal is negated. The MC68030's memory management unit may be disabled using the MMUDIS* input to the 68030. Access to this signal is provided using switch S4.

3.3   Memory Module

This section describes the operation of the RAM and ROM modules. The RAM architecture is based upon a single thirty-two bit wide bank of 70 ns static column RAM (SCRAM) with a capacity of one megabyte. The SCRAM controller is based upon a high density PLD, the AMD Mach 110, with high resolution timing generated by

the AMD Am2971A programmable event generator (PEG). The ROM architecture is based upon a single eight-bit wide bank of EPROM with a capacity of 128 kilobytes. The ROM is only intended for SCSI control processor diagnostic and operating code. Time critical code sections are moved from the ROM to the main memory, the SCRAM. Microcode and data may be loaded from the host after boot. In situations where the InvestiGATOR is being used as a standalone data collection unit microcode might be loaded from a non-volatile semiconductor disk resident on the I/O bus.

3.3.1 Static Column RAM

The InvestiGATOR contains a one megabyte bank of SCRAM. The SCRAM is used as an alternative to standard DRAM because of its high speed access properties: sequential accesses to the same column proceed substantially faster than an access to the same speed rated standard DRAM. The SCRAM achieves no-wait-state operation when operating in static column mode. This is an attractive property when coupled with the 68030's burst mode and when one considers that the primary use for this bank of RAM will be to perform SCSI block transfers.

There are penalties to pay for the high performance of the SCRAM: SCRAM is fifty per cent to one-hundred per cent more expensive than standard DRAM, SCRAM requires significantly more control logic than standard DRAM, and in the event of a non-static column mode access, there is a substantial penalty to pay in cycling a new row address. However, given the design constraints, the static column architecture is the best solution.

The SCRAM architecture is composed of several components. There is the SCRAM itself, data transceivers, address multiplexer, address comparator, burst counter, refresh timer, high-time resolution sequencer, and byte select decoder. A

block diagram of the SCRAM architecture is shown in Figure 3.4.
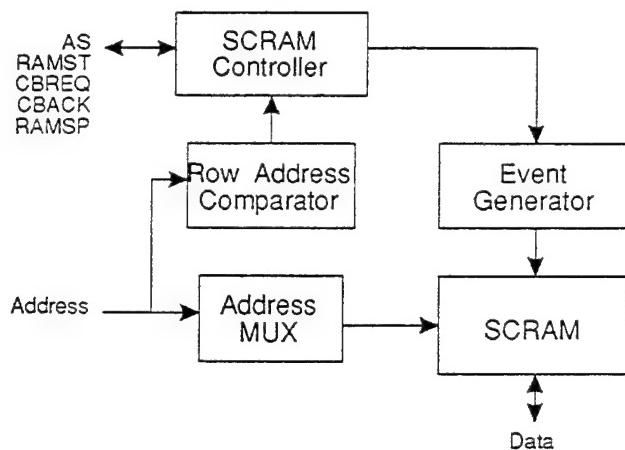
Figure 3.4: Block Diagram of SCRAM Architecture

The burst counter serves to cycle the two lowest order bits of the address during burst accesses. For example, if an access is a miss in the 68030's internal cache, caching is allowed, and the target of the access supports burst mode accesses then in order to keep latency (from the execution unit's point of view) minimal the required word is read. Then the next longword address, modulo four, is read, and so on until four longwords have been read. The burst counter is integrated onto the PLD which contains the controller state machine.

The address comparator serves to allow the controller to determine whether an access is a static column hit. The address comparator contains both a register and a comparator so that the previous row address can be stored for comparison with future accesses. Note that refresh cycles do not invalidate the register contents of the address comparator. Validity of the contents of the address comparator is controlled by the state of the RAS signal: the contents (and thus the output) of the address comparator are valid if and only if RAS is asserted.

The refresh counter is a simple eight bit counter whose trickle-count output

sets a refresh request to the controller state machine. The refresh counter issues a refresh request 256 cycles ($T=50$ ns) after it is reset for a net of one request every 12.8 ms resulting in each of the 512 rows of RAM being refreshed every 6.6 ms, meeting the required 8 ms refresh cycle period.

The controller issues commands to the sequencer to perform operations on the RAM. The sequencer is an AMD Am2971A programmable event generator (PEG) which is capable of generating sequences of signals with 10 ns timing resolution. Some of the signals are routed directly to their targets while others are routed through a PLD which provides byte select coding, primarily for write operations. Additionally the controller handles all handshaking with the CPU. The state machine must handle a number of conditions:

- Refresh cycle

- Static column miss, read without burst

- Static column miss, read with burst

- Static column hit, read without burst

- Static column hit, read with burst

- Static column miss, write

- Static column hit, write

Examining the controller state machine diagram (see Figure B.3) we see that the state machine implements the read sequences using a variety of shared state sequences. By sharing state sequences we arrive at a much more efficient implementation of the controller state machine.

The following refers to Figure 3.5. The static column RAM device is dependent upon four control signals: chip select (CS), row address strobe (RAS), write strobe (WR), and output enable (OE). RAS, WR, and OE are generated by the PEG and fed directly to the SCRAM devices while the CS signal is generated by the PEG it is subject to byte select coding by PAL4 using the byte select signals (UU, UM, LM, LL) generated by the CPU module. The data lines are buffered using four Am29C861A CMOS bus transceivers under the control of the SCRAM controller state machine. The address lines are multiplexed by a pair of Am29C827A bus drivers acting as a row/column address multiplexer under control of the controller via the PEG.

1: Address Bank Decoder
2: Byte Select Conditioned Chip Selects
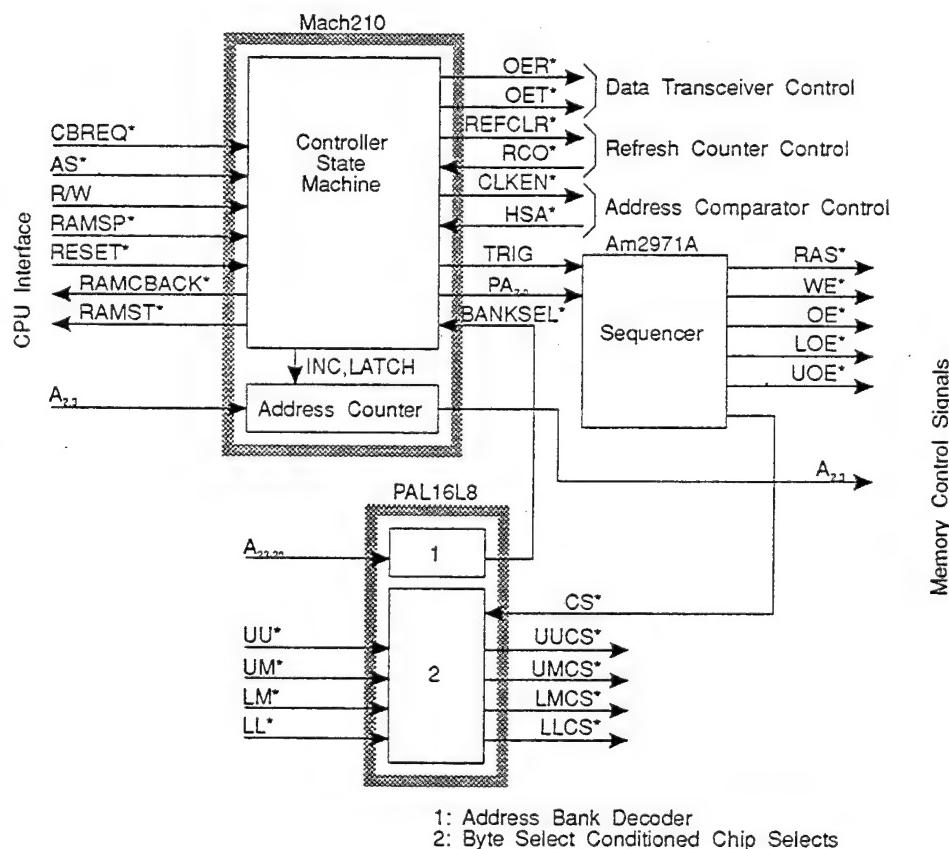
Figure 3.5: SCRAM Controller Architecture

The refresh counter operates in a free counting mode, driven by the 20 MHz

system clock. Two-hundred fifty-six clock cycles (12.8 ms) after a counter reset the trickle-count output (RCO) is asserted for one clock cycle which in turn sets the refresh request SR flip-flop in the controller state machine PLD. After the completion of the current memory transaction the controller resets the counter and the refresh request SR flip-flop via the CLRREF signal and orders the PEG to execute a hidden refresh cycle. Under worse case conditions a refresh request could suffer a response latency of up to twelve clock cycles (600 ns). Thus, under these worse case conditions a hidden refresh cycle might be executed every 13.4 ms implying a refresh of every row of the SCRAM every 6.9 ms, still within the required 8.0 ms.

The controller handshakes with the CPU module via the AS, CBREQ, RAMSP, Read/Write, CBACK, and STERM signals. The R/W, AS and RAMSP signals are used in conjunction with additional address decoding provided by PAL4 (via the BANKSEL signal from PAL4) to initiate memory transactions. The CBREQ/CBACK handshaking pair is used to control burst cycles.

The controller orders the PEG to execute sequences using the PA2-0 and TRIGx outputs. The PA2-0 signals provide an address to the PEG to determine the starting point in its memory for execution while the TRIGJ/TRIGK outputs are fed through a negative- edge triggered flip-flop to generate a trigger signal which will arrive at a time when the PEG address inputs (PA2-0) are guaranteed valid and cause the PEG to begin execution with minimal latency. The chip select signals generated by the PEG are gated using the byte selects generated by the CPU module with controller override via the CSALL signal. The PEG also generates the RAS, WR, and OE signals used by the SCRAM. Additionally, the PEG controls the address multiplexer via the AREG signals which control the output enables of the address drivers.

The address comparator is a combination register/comparator. The controller causes the comparator to latch a new row address using the CLKEN signal. When the address comparator determines that the row address at its input matches that stored in its internal register it signals the controller using the HSA signal. Finally, the data transceivers are controlled by the OER and OET controller signals.

The burst address counter is integrated into the controller PLD. This counter is a simple two-bit counter with load and increment controls from the controller state machine, load inputs AI1,0, and AO1,0. Negation of the load or latch and increment controls implies a hold state. The outputs of this counter are fed through the address multiplexer to the SCRAM array. Note that since the least significant bits of the column address are fed through the burst address counter, the presentation of a new column address to the SCRAM array is limited by both the address multiplexer and the speed with which the address counter can latch a new address and present it to the address multiplexer.

The SCRAM must be verified each time the power is applied. There are standard algorithmic test methods which facilitate functional testing of the DRAM and detection of common faults [6]. The standard test methods discussed in [6] are targeted primarily at functional testing of DRAMs in VLSI testers, not testing of the memory in circuit. These methods may be adapted with the addition of tests to exercise the surrounding architecture. In particular, during testing of the first InvestiGATOR board, a stuck-at fault (SAF) was discovered in one of the address multiplexer buffers. A test to find SAFs in the address multiplexer buffers is given in Figure 3.6. Once the address multiplexers are verified the data transceivers should be verified. Note that malfunctioning data transceivers could potentially mask or simulate an address multiplexer SAF, thus, special precaution should be taken in

the implementation of the address multiplexer SAF detection so as not to cause an erroneous conclusion as to the status of the address multiplexers.

```
for i=0 to n-1
  M[0]:=0
  M[2^i]:=1
  if M[0]!=0 then there exists an SA0 fault @ bit i
  M[0]:=1
  M[2^i]:=0
  if M[2^i]!=1 then there exists an SA1 fault @ bit i
end
```

Figure 3.6: Pseudo-Code for Address Multiplexer SA Fault Detection

Once the status of the surrounding architecture is verified, [6] suggests that tests for unlinked SAFs, unlinked transition faults (TFs), unlinked coupling faults (CFs), linked CFs, linked CFs and TFs, address decoder faults (AFs), and various pattern sensitive faults (PSFs) be conducted. It turns out that two tests will provide fault coverage for SAFs, TFs, AFs, linked CFs, linked TFs, unlinked idempotent, and unlinked inversion CFs: the March C and March B algorithms.

Each march element of a march sequence consists of an arrow pointing up or down, indicating the direction of march in address space, and a sequence of read and write operations. For example, $\Uparrow$ indicates an address sequence from zero to $n - 1$, while $\Downarrow$ indicates an address sequence from $n - 1$ to zero. The March C algorithm is given in Figure 3.7. The March B algorithm is given in Figure 3.8. Both the March C and March B algorithms assume that an initial $\Uparrow(w0)$ march is executed to initialize the memory before the test algorithm is executed.

The most common PSFs which occur are neighborhood pattern sensitive faults (NPSFs). NPSFs are faults where the writing of memory cells adjacent to a base cell will cause an unwanted transition in the base cell. The cells most likely to effect a

$$\{ \Uparrow(r,w1); \Uparrow(r,w0); \Uparrow(r); \Downarrow(r,w1); \Downarrow(r,w0); \Downarrow(r); \}$$

Figure 3.7: March C Algorithm for Memory Testing

$$\{ \Uparrow(r,w1,r,w0,r,w1); \Uparrow(r,w0,w1); \Downarrow(r,w0,w1,w0); \Downarrow(r,w1,w0); \}$$

Figure 3.8: March B Algorithm for Memory Testing

base cell – and thus expose an NPSF – are the four cells adjacent to the base cell in the north, south, east, and west directions. A basic NPSF detection algorithm, suggested by [6] is given in Figure 3.9.

```
write all base cells with zero;
for each base cell
  apply a pattern;
  read base cell and compare against expected value (zero);
end;
write all base cells with one;
for each base cell
  apply a pattern;
  read base cell and compare against expected value (one);
end;
```

Figure 3.9: A Basic NPSF Detection Algorithm

3.3.2 ROM Controller and Architecture

The InvestiGATOR contains a single bank of 128K × 8-bit wide (128 kilobytes) EPROM. This ROM is a low performance memory which contains basic firmware for the InvestiGATOR and may contain some firmware for the array under test. ROM read cycles are executed in three clock cycles yielding a net bandwidth of 6.67 megabytes per second. Code segments demanding higher performance may be

shadowed to the RAM space.

3.4   I/O Bus and Devices

The InvestiGATOR supports an I/O bus through which it communicates with the out-
side world. Currently the I/O bus contains a SCSI controller, and a serial (RS-232C)
port. The SCSI controller utilizes the Western Digital 33C93A SCSI bus controller
chip and contains a thirty-two kilobyte data buffer. The serial I/O controller uses the
AMD Z85C30 ESCC (Enhanced Serial Communications Controller) to provide two
channels of RS-232 I/O. Allowances are made for the addition of peripherals to the In-
vestiGATOR's I/O bus. Some of the allowances include a wired-OR interrupt request
line and three data transfer acknowledge lines: one for each size data port supported
by the MC68030. The accessibility of the I/O bus is intended to compensate for the
potential unavailability or unsuitability of a SCSI bus equivalent peripheral.

3.4.1   SCSI

The SCSI port is built around the Western Digital 33C93A SBIC (SCSI Bus Interface
Chip). The SCSI port is designed to use a form of I/O called DBA (direct buffer
access) for data block transfers. Using DBA, the SBIC performs block transfers
directly to and from a thirty-two kilobyte local buffer memory without processor
intervention. This allows the SBIC to achieve its rated five megabyte/second data
transfer rates and allows the control processor to avoid the performance penalties
associated with interrupt servicing overhead. A block diagram of the SCSI port
architecture is depicted in Figure 3.10.

The SBIC operates in two modes during normal operation in the InvestiGA-
TOR: direct addressing mode and DBA mode. In the direct addressing mode the
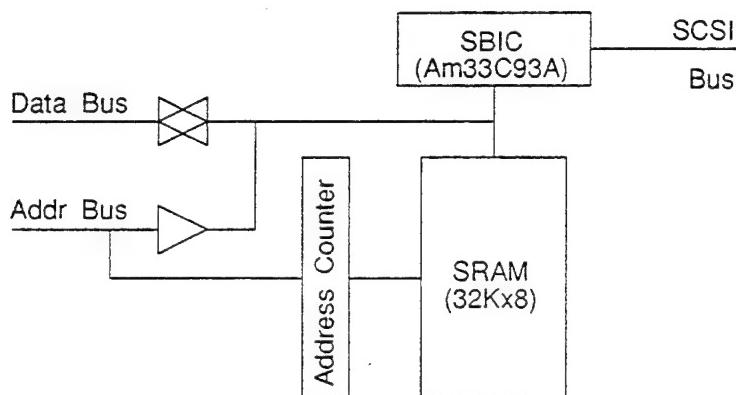
Figure 3.10: Block Diagram of the SCSI Port

processor performs transactions with the SBIC by using hardware assisted time multiplexing of the address and data to the SBIC address/data port. Direct addressing mode contrasts with indirect addressing mode where the processor first would write an address to the SBIC and then the next SBIC access would be performed on the register whose address was written in the previous cycle. Indirect addressing mode carries obvious penalties since two real accesses are required for every data transaction. The SBIC normally is kept in a DBA stand-by mode: that is, whenever the processor is not accessing the SBIC or RAM buffer the SBIC is in DBA mode. When the processor attempts to perform a transaction with the SBIC or RAM the SBIC is switched out of DBA mode so that the transaction may proceed.

In DBA mode the SBIC has control of the RAM buffer. Reads and writes are accomplished using the SBIC read enable and write enable signals. Since the SBIC has no means of handshaking with external logic when performing individual transactions with the buffer RAM, it is up to the control architecture to ensure that the transaction meets the SBIC's timing requirements. Additionally, the SBIC provides no direct control of the address counter; rather, the control of the counter is implicit. After each buffer read or write operation, the counter must be incremented

by the external hardware. The control logic determines when to increment the counter by observing the read and write strobes. Address counter control in DBA mode is performed by observing the RE and WE strobes which are controlled by the SBIC in this mode.

The SCSI-2 specification gives a list of commands which a processor on the SCSI bus can implement. Some of the commands listed are optional while others are mandatory under the SCSI-2 specification. A table of these commands and whether the InvestiGATOR responds to the commands is given in Table 3.3.

	Command Name	Notes
O	Change Description	Not Implemented.
O	Compare	Not Implemented.
O	Copy	Not Implemented.
O	Copy and Verify	Not Implemented.
M	Inquiry	
O	Log Select	
O	Log Sense	
O	Read Buffer	Used to read program memory and control store.
O	Receive	Used to transmit command and data packets to InvestiGATOR.
O	Receive Diagnostic Results	Used to retrieve diagnostic results.
M	Request Sense	
M	Send	Used to receive command and data packets from InvestiGATOR.
M	Send Diagnostic	Used to request diagnostics to be performed.
M	Test Unit Ready	
O	Write Buffer	Used to load program memory and control store.

O=optional, M=mandatory, according to SCSI-2 definition.

Table 3.3: SCSI-2 Command Set

3.4.2 SIO

The serial I/O interface is provided for software development and diagnostic purposes. The serial controller is based upon an AMD Z85C30 Enhanced Serial Communications Controller (ESCC). The ESCC-I/O bus interface is composed simply of an eight-bit buffer and a PAL-based controller.

The ESCC supports two channels of serial communications and independent baud rate generation. Two channels of serial I/O are supported by the InvestiGATOR since the additional cost is minimal. In the case of the InvestiGATOR the baud rate is generated by dividing down the 10 MHz system clock to the appropriate baud rate. The baud rate is programmed by providing a time constant for each channel. The time constants appropriate to some common baud rates assuming f_{CLK}=10 MHz, and a clock multiplier of sixteen are provided in Table 3.4.

Desired Baud	Time Constant	Actual Baud	Per Cent Difference
300	1044	299.904	-0.032
1200	262	1201.92	0.159
2400	132	2403.85	0.158
4800	67	4807.69	0.155
9600	35	9469.70	-1.296
19200	18	19531.3	1.510

Table 3.4: Time Constants versus Baud Rates for Enhanced Serial Communication Controller

The ESCC's registers are mapped in I/O space as described in Table 3.5.

The serial ports are brought out to DB9 connectors on the back of the InvestiGATOR. The signals are translated via the RS-232C level compatible MC1448 transmitter and MC1449 receiver. This transmitter/receiver pair was chosen for its robustness. The pinout of the InvestiGATOR's serial ports is non-standard and de-

Address	Description
20800h	Channel B Control Register.
20801h	Channel B Data Register.
20802h	Channel A Control Register.
20803h	Channel A Data Register.

Table 3.5: Enhanced Serial Communication Controller Register Memory Map

picted below in Figure 3.11.
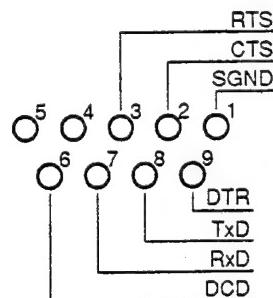
Figure 3.11: InvestiGATOR Serial Port Pinout

A cable suitable for connecting the InvestiGATOR to an IBM PS/2 host was constructed according to the diagram in Figure 3.12. The cable is suitable for XON/XOFF flow-control protocol and is not suitable for hardwire (*i.e.*, REQ/ACK or RTS/CTS) protocols. Note that the InvestiGATOR end of the cable does not have the usual data set ready (DSR) and ring indicator (RI) inputs. Furthermore, the InvestiGATOR does not offer a protective ground (PGND) input. The protective ground wire from the terminal side of the cable should be connected and provide grounding for the cable shielding. However, the signal ground (SGND) is connected.

3.5   I/O Expansion

The I/O expansion connector is intended to allow unforeseen problems to be addressed. The I/O expansion connector is mapped to the to I/O address space and

36
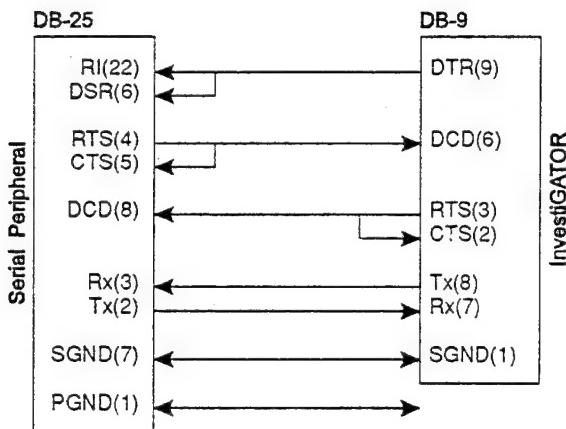
Figure 3.12: InvestiGATOR to IBM PS/2 Serial Cable

may be used with eight, sixteen, and thirty-two bit data bus sizes. Wired-OR lines are provided for asynchronous bus cycle termination and interrupts. The port is fully buffered and the address lines and control lines are always turned on, thus allowing the I/O expansion connector to be used to probe system activity. A list of signal names, pin numbers, and description of the signals' functions are given in Table 3.6.

Pin Number	Signal Name	Description
0—31	D31–0	Data bus.
32—51	A19–0	Address bus.
52	AS*	Address strobe.
53	DS*	Data strobe.
54	IOSP*	I/O address space flag.
55	IO8DTACK*	Eight bit port DTACK.
56	IO16DTACK*	Sixteen bit port DTACK.
57	IO32DTACK*	Thirty-two bit port DTACK.
58	IOIRQ*	I/O expansion port IRQ line.

Table 3.6: I/O Expansion Connector Signals

3.6 Array Bus

The array bus is a connection rich environment. Previous experience and analysis has led to the conclusion that interboard connectivity was lacking in traditional host environments such as the PC-XT, PC-AT, EISA, MicroChannel, VME, and others.

The InvestiGATOR has a 324 signal connector. Seventy-five of the signals on this bus are allocated for a memory mapped interface to the MC68030 SCSI control processor. These signals are fixed in terms of arrangement and function. The remaining signals are broken up between near-neighbor connections and broadcast connections which are functionally undedicated *a priori*. One-hundred forty of these signals are wired as near-neighbor connections where seventy of the signals go to the right adjacent slot and the remaining seventy go to the left adjacent slot. The remaining one-hundred nine signals are wired as a broadcast bus to the array. All of the near-neighbor connections are array broadcast connections are invisible to the MC68030 CPU. A breakdown of the allocation of these signals is listed in Table 3.7.

3.6.1 CPU to Array Bus Interface and Architecture

The CPU is interfaced to the array bus via a memory mapped interface using a total of seventy-five signal lines on the backplane connector. The interface to the array bus buffers the CPU signals and passes all signals necessary for data and instruction transactions to take place. A breakdown of the allocation of these signals is listed in Table 3.7.

This interface does not support alternate address spaces via the 68030's function code (FCx) outputs, dynamic bus sizing (*i.e.*, all ports are thirty-two bits), nor does it support burst mode accesses. Each slot has its own STERM signal which is routed to the CPU by the interface. STERM validity is ascertained by observation

of the SLOTENx signals. Each slot has a wired-OR SLOTEN (active-low) signal which is held high if a card is not present in a slot. If a card is present and needs to be able to assert STERM then it must assert the SLOTENx signal by wiring the signal directly to ground. The STERMx and SLOTENx signals are unique at each connector and are hidden from the other slots.

The array bus error (ARYBERR) and interrupt request (ARYIRQ) signals are wired-OR. ARYBERR causes a BERR cycle to be executed by the 68030, while ARYIRQ requests a level one priority 68030 IRQ.

3.6.2 Local (Near-Neighbor) Connections

The local slot connections consist of seventy signal lines to each adjacent slot. While these connections are not predefined, they are adequate to implement a sixty-four bit, bidirectional communication port or a pair of thirty-two bit unidirectional ports to each adjacent slot. These signals are unused in the Gauss machine implementation, but will be used in a future TMS320C40 hypercube implementation.

3.6.3 Array Broadcast Bus

The array broadcast bus consists of the remaining 109 signal lines not used in the near-neighbor connections or the CPU-array interface. Like the near-neighbor connections, the broadcast connections are not defined *a priori*. These connections are intended to handle control and data distribution. The assignment of these signals for the Gauss machine is discussed in Section 6.2.

3.7 Support Circuitry

This section describes the miscellaneous modules that provide the critical support functions which are not a proper part of any of the major modules of the architecture.

3.7.1 Clock Generator Module

The clock generator module consists of three components: the crystal time base, the clock generator, and a low-skew buffer. The crystal timebase is a 40 MHz TTL compatible clock. This clock drives an AMD Am2971A PEG (Programmable Event Generator) which produces phase locked versions of 2 MHz, 5 MHz, 10 MHz, and 20 MHz clocks. Finally, since the PEG has a relatively low power output drive, the clock signals are buffered by an AMD Am29C827A high-speed CMOS bus driver. The Am29C827A features low t_{PD}, low skew, and "edge-rate control" which is intended to minimize ground bounce.

The clock module produces one copy each of the 2 MHz and 5 MHz clocks, two copies of the 10 MHz clock, and six copies of the 20 MHz clock. The various copies of the 20 MHz clock are reserved for distribution to different modules, with the intent of minimizing clock skew within each module. The clock distribution reservation table is shown in Table 3.8.

3.7.2 Reset Circuit Module

The reset circuit module contains power-up and on demand system reset circuitry. Power-up reset is provided by a Texas Instruments TL7705A Power Supply Supervisor/Reset Generator. The power-up reset circuit monitors system power and asserts the RESET signal for an amount of time controlled by C1. C1 has been chosen to be greater than $40\mu F$, thus, RESET will be asserted for at least 500 ms after the 5V

supply rail reaches within ten per cent of 5V.

The reset signal provided by the TL7705A is buffered into the wired-OR system RESET* signal by an open-collector inverter. The reset circuit contains a reset switch connected to the system RESET* signal.

Pin Number	Signal Name	Description
1	SLOTENx*	Slot enable. Wired-OR.
2	STERMx*	Synchronous bus cycle termination.
3	ARYDS*	Data strobe.
4	ARYAS*	Address strobe.
5	ARYR/W	Read/write strobe.
6	ARYUU*	Upper byte select.
7	ARYUM*	Upper-middle byte select.
8	ARYLM*	Lower-middle byte select.
9	ARYLL*	Lower byte select.
10	ARYARYSP*	Array address space select.
11	ARYRMC*	Read-modify-write signal.
12	RESET*	System reset.
13	HALT*	System halt.
14—45	D31-0	Data bus.
46—75	A29-0	Address bus.
79	CLK20C	20 MHz system clock.
81	CLK10B	10 MHz system clock.
83	CLK5	5 MHz system clock.
85	CLK2	2.5 MHz system clock.
77,82,84,87	Vcc	5 V power bus.
76,78,80,86	GND	Ground rail.
88—?	—	Near neighbor connections. Odd pin numbers to left slot. Even pin numbers to right slot.
?—324	—	Broadcast bus.

Table 3.7: Array Bus Signals

42

Signal	Frequency	Reservation/Availability
CLK2	2 MHz	Unallocated
CLK5	5 MHz	Unallocated
CLK10a	10 MHz	I/O module
CLK10b	10 MHz	Array module
CLK20a	20 MHz	CPU module
CLK20b	20 MHz	I/O module
CLK20c	20 MHz	Array module
CLK20d	20 MHz	RAM module
CLK20e	20 MHz	ROM module
CLK20f	20 MHz	Unallocated

Table 3.8: Clock Reservation

:

Chapter 4

SOFTWARE

The InvestiGATOR's firmware is written primarily in C. Besides being readily available for the 68030 architecture, the C language offers high level language benefits of compactness and ease of use combined with some of the benefits associated with assembly language, mainly control and speed. The InvestiGATOR firmware is modular in nature, composed of a kernel, SCSI bus interface (SBIC) firmware, serial I/O (SIO) firmware, and interface code to the target processor, the Gauss machine. A block diagram of the software architecture is shown in Figure 4.1.
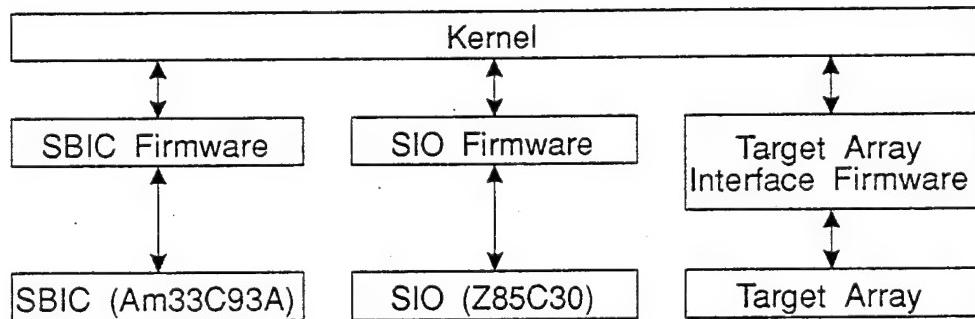
Figure 4.1: InvestiGATOR Software Architecture Block Diagram

4.1 Kernel

The primary mission of the kernel is to manage resources and control dispatch of tasks to the various subsystems. The key resource which is managed by the kernel is memory. The kernel also manages the dispatch of interrupts to the various

subsystems.

4.2   SBIC Firmware

The SBIC firmware is responsible for managing the substantial SCSI protocol. The following sections introduce the operation of the SCSI bus and the structure of the SBIC firmware.

4.2.1   SCSI Bus Operation

The SCSI bus has four phases of operation. The SCSI bus idles in the bus free phase. When a device wants to gain control of the bus, the bus enters the arbitration phase. During the arbitration phase all devices attempting to gain control over the bus arbitrate for the bus. The device with the highest SCSI ID wins the arbitration. After successful arbitration the bus enters the selection phase. During the selection phase the SCSI bus master attempts to select the device with which it wants to communicate. After successful selection the bus enters the information transfer phase. The information transfer phase is characterized by the transfer of commands, data packets, and messages. A flow diagram of the SCSI phases is shown below in Figure 4.2.

There are two types of devices on the SCSI bus: initiators and targets. Initiators are typically host processors while targets are typically peripheral devices such as disk drives. The InvestiGATOR operates as a target. The InvestiGATOR responds to the commands test unit ready, request sense, send and receive. These operation of the these commands are shown in Figures 4.3–4.6.

The test unit ready command is used to query the target device as to its status. This command is mandated by the SCSI standard. The InvestiGATOR will
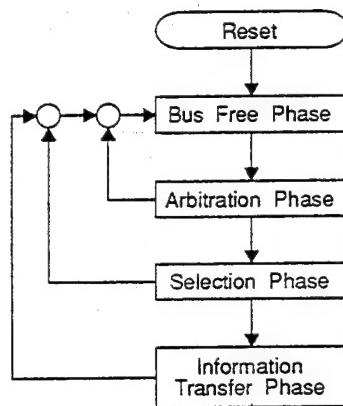
Figure 4.2: SCSI Bus Phases

respond with a good, check condition, or busy status code. The good status code indicates that the InvestiGATOR is ready and standing by for a command. The check condition status code indicates that the InvestiGATOR is not ready and has additional status information available. Finally, the busy status code indicates that the InvestiGATOR is busy. The transactions required to execute a test unit ready command are shown in Figure 4.3.
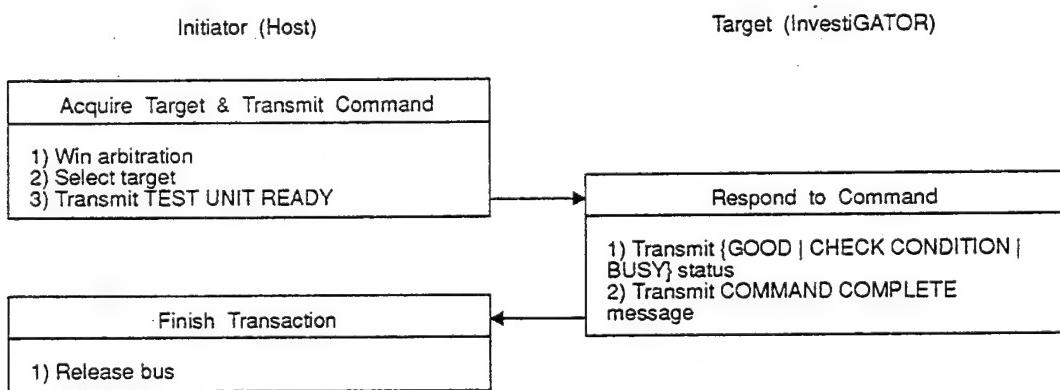
Figure 4.3: Test Unit Ready Command Operation

The request sense command is used to query the device for extended status data. Typically, the request sense command is executed after a check condition status

is returned on a command. The transaction model for the request sense command is shown in Figure 4.4.

Initiator (Host)   Target (InvestiGATOR)

```
┌─────────────────────────────────────────┐
│ Acquire  Target  &  Transmit  Command    │
├─────────────────────────────────────────┤
│ 1) Win arbitration                        │
│ 2) Select target                          │        ┌──────────────────────────────────────────┐
│ 3) Transmit REQUEST SENSE                 │───────▶│ Respond  to  Command                      │
└─────────────────────────────────────────┘        ├──────────────────────────────────────────┤
                                                     │ 1) Enter DATA IN phase                    │
                                                     │ 2) Transmit sense data                    │
┌─────────────────────────────────────────┐        │ 3) Transmit GOOD status                   │
│ Finish  Transaction                       │        │ 4) Transmit COMMAND COMPLETE              │
├─────────────────────────────────────────┤◀───────│ message                                   │
│ 1) Release bus                            │        └──────────────────────────────────────────┘
└─────────────────────────────────────────┘
```
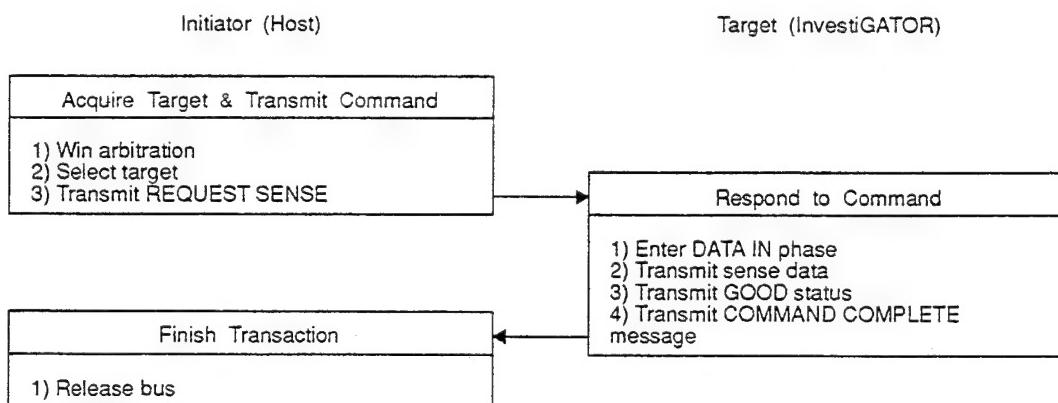
Figure 4.4: Request Sense Command Operation

The send and receive commands are the primary data communication commands between a host processor and the InvestiGATOR. The transaction models for the send and receive commands are shown in Figure 4.5 and Figure 4.6.

Initiator (Host)   Target (InvestiGATOR)

```
┌─────────────────────────────────────────┐
│ Acquire  Target  &  Transmit  Command    │
├─────────────────────────────────────────┤
│ 1) Win arbitration                        │
│ 2) Select target                          │        ┌──────────────────────────────────────────┐
│ 3) Transmit SEND                          │───────▶│ Respond  to  Command                      │
└─────────────────────────────────────────┘        ├──────────────────────────────────────────┤
                                                     │ 1) Enter DATA OUT phase                   │
                                                     │ 2) Transmit data                          │
                                                     │ 3) Transmit {GOOD | CHECK CONDITION |     │
┌─────────────────────────────────────────┐        │ BUSY} status                              │
│ Finish  Transaction                       │◀───────│ 4) Transmit COMMAND COMPLETE              │
├─────────────────────────────────────────┤        └──────────────────────────────────────────┘
│ 1) Release bus                            │
└─────────────────────────────────────────┘
```
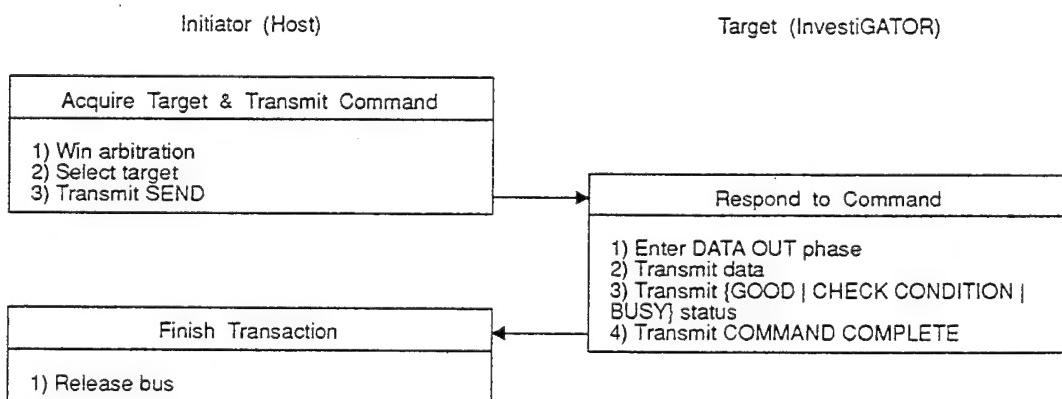
Figure 4.5: Send Command Operation

Initiator (Host)   Target (InvestiGATOR)

Acquire Target & Transmit Command
1) Win arbitration 2) Select target 3) Transmit RECEIVE

Respond to Command
1) Enter DATA IN phase 2) Receive data 3) Transmit {GOOD | CHECK CONDITION | BUSY} status 4) Transmit COMMAND COMPLETE

Finish Transaction
1) Release bus
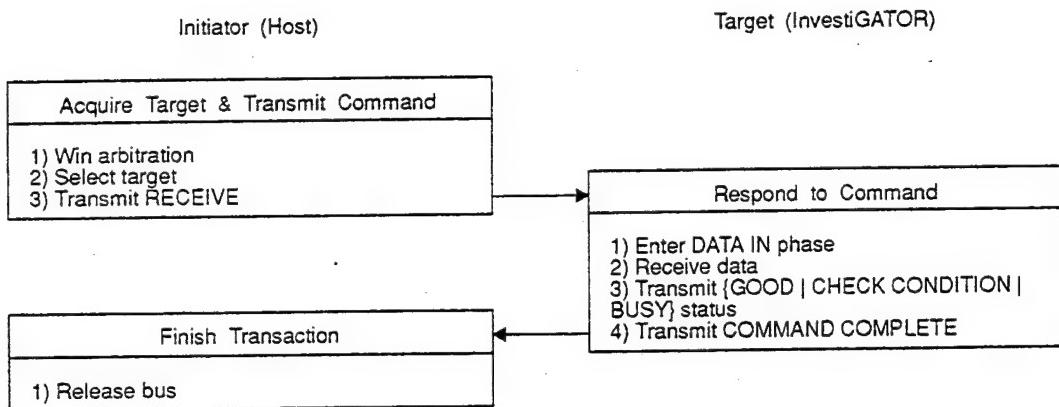
Figure 4.6: Receive Command Operation

4.2.2 SBIC Firmware

The SBIC operates under an interrupt driven protocol. This subsection discusses the flow diagram of the SBIC reset routine and the interrupt service routine (ISR) depicted in the flow diagram of Figure 4.7.

Before the SBIC can be used, it must be initialized via a software interrupt. The SBIC is preloaded with the SCSI address of the InvestiGATOR before a software reset is executed. After the reset completes, interrupts and data I/O modes are programmed. Initially, the InvestiGATOR is set to SCSI address 4 and uses interrupt drive I/O.

The InvestiGATOR operates only as a target in the initial configuration. The InvestiGATOR does not support disconnect/reselection at this time so the firmware is fairly simple. The SBIC interrupts the processor with a service required interrupt when an initiator on the SCSI bus selects the InvestiGATOR. Selection may occur either with the attention (ATN) signal asserted or negated: ATN asserted indicates that there is a message pending. Selection with attention is used exclusively to request that the target accept an IDENTIFY message. The InvestiGATOR does not

currently support the IDENTIFY message, and thus selection with attention leads to a fault condition.

After the SBIC ISR identifies a selection without attention condition, the ISR prepares the SBIC to receive a command from the initiator. Currently the InvestiGATOR only supports a SCSI command set which is (coincidently) limitted to those commands which have six byte command frames. Thus, a transfer count of six is loaded into the transfer count register and a RECEIVE COMMAND command is issued to the SBIC. The SBIC then receives a command from the initiator.

If a data phase is required by the command received from the initiator then the SBIC is prepared for a data phase by setting the synchronous transfer control register and the transfer counter register and issuing a send or receive data command.

If the command received was a linked command then a SEND STATUS command is issued to the SBIC and the execution returns to the RECEIVE COMMAND phase. If the command was not a linked command then a SEND STATUS AND COMMAND COMPLETE command is issued to the SBIC, causing the last command's status to be transmitted to the initiator and the SBIC to disconnect.

4.3 SIO Firmware

The serial port is operated in an interrupt driven I/O mode. The SIO drivers support circular transmit/receive buffers which aid in increasing system throughput and allowing type-ahead. The XON/XOFF flow control protocol is the only flow control protocol currently supported. In the current implementation serial port A is the console (stdin/stdout) while serial port B is unassigned.
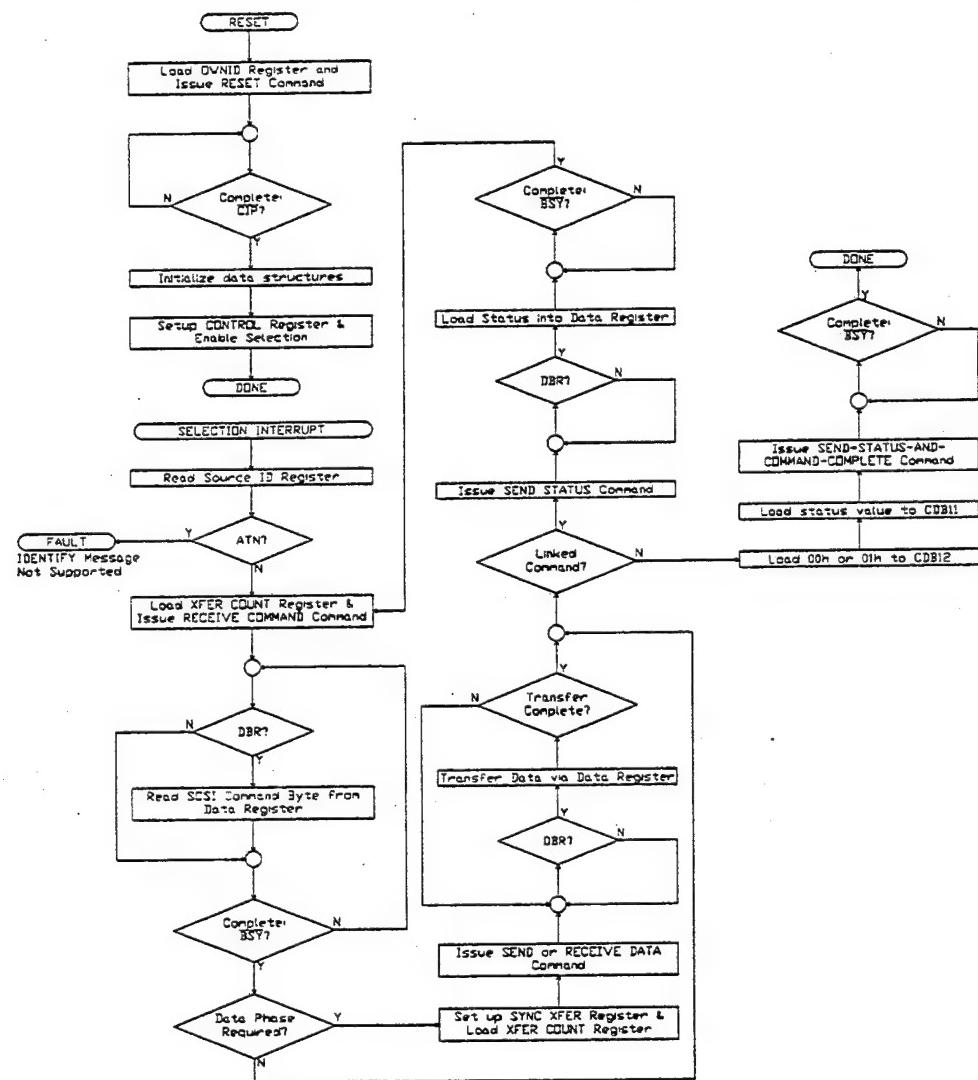
Figure 4.7: SBIC Interrupt Service Routine Flow Diagram

Part III

Gauss Machine

Chapter 5

INTRODUCTION

The Gauss machine is a 2×2 systolic array processor comprised of three seven-bit GEQRNS channels for a total of six seven-bit RNS channels. The array of processors is arranged in a mesh-connected topology with unidirectional dataflow. Alternately, the Gauss machine may be configured to utilize two of its processors as a vector processor. The Gauss machine excels in computation of level 3, level 2, and level 1 operations.

5.1 Motivation

The design of the Gauss machine is motivated by several factors. There exists a need for high-performance front-end signal processors which are reliable, small, consume minimal power, and are relatively inexpensive. Typically, high performance is achieved using a combination of fast processors coupled with some parallelism. Signal processing applications have been demonstrated to be particularly amenable to systolic array implementations[7]. Traditional technologies have typically featured large, multiple package designs where individual processors were made up of several large VLSI devices[8]. Even new, state-of-the-art processors designed for parallel processing, but based on conventional arithmetic technology such as the iWarp[9] or TMS320C40[10] have at least one large package per processor element. These designs typically had large physical form factors, high power consumption (multiple watts per processor), and low reliability. Attempts to improve reliability by incorporat-

ing redundancy typically result in little improvement at the expense of greater than one-hundred per cent in terms of hardware, power, size, and cost.

Processor architectures based upon residue arithmetic are uniquely qualified to meet the demanding needs of modern signal processing systems. The RNS is a high performance system of arithmetic having performance which is independent of word-width. The RNS features relatively small die area when compared with conventional arithmetic. The RNS is inherently fault and defect tolerant[3, 4], and may realize the full potential of VLSI systolic arrays[7].

5.2 Design Parameters

Currently, there are no RNS systems which are general purpose in nature. Most RNS systems are hard-wired to a specific task. There exists a need to demonstrate an RNS system which is more general purpose in nature. This RNS system must be capable of many different operations. Additionally, there is motivation to demonstrate the use of the RNS in systolic array architectures.

The Gauss machine is designed as a discrete prototype of a $2 \times 2 \times 6$ VLSI systolic array of GEQRNS multiplier-accumulators. The array is hosted by the InvestiGATOR array processor testbed. Data conversion functions are provided by the InvestiGATOR. The array controller is a microprogrammed controller based upon a single chip microsequencer.

Chapter 6

IMPLEMENTATION

6.1   Architecture

The Gauss machine supports a three channel GEQRNS or QRNS, 2×2 array of seven bit multiplier-accumulators. The array is formed by six boards, with each board comprising a 2×2 array seven bit multiplier-accumulators. The array is integrated into the InvestiGATOR array processor backplane with the addition of a controller, and optionally, a forward-conversion and CRT engine board.

The Gauss machine supports a mesh connected geometry with north and east flow of data. The array uses FIFOs to provide the means for data to be sequenced through the array. The FIFOs are the gateway through which the array communicates with the outside world. Additionally, the Gauss machine offers a vector mode of operation which utilizes PEs (1,1) and (1,2) to perform level 1 and level 2 operations at higher performance levels than would be possible using the full array. A block diagram is given below in Figure 6.1.

The FIFOs located on the periphery of the array meet the goal of allowing concurrency in processing and data I/O since the memories may be loaded or emptied as calculations proceed.

54
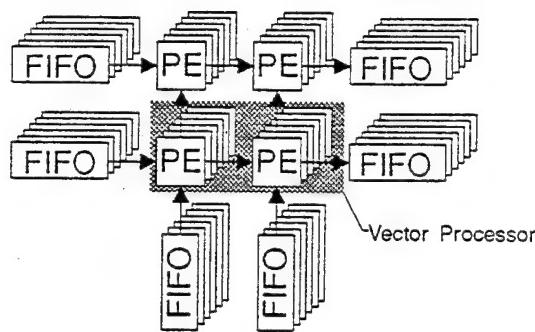
Figure 6.1: Block Diagram of Gauss Machine Array

6.2   Processor Implementation

Each processor element in the array (see Figure 6.1) consists of a multiplier, accumulator, and support architecture. The inputs to the multiplier come from the X-bus and Y-bus. The X-bus is also connected to the F-bus, allowing the accumulator to be pre-loaded, or the output of the adder may be output to the X-bus. A block diagram of the processor element is depicted below in Figure 6.2.
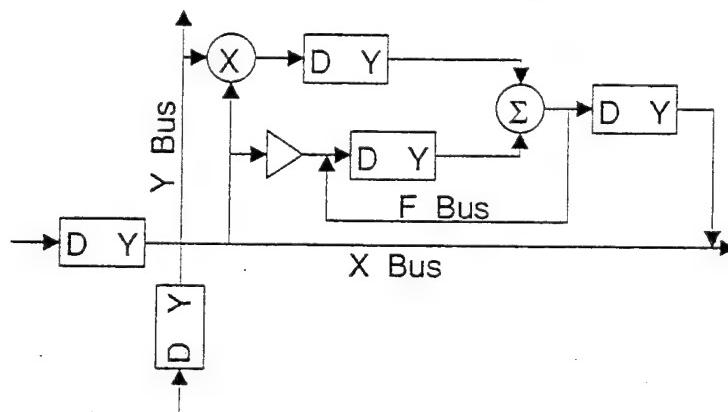
Figure 6.2: Block Diagram of Gauss Machine Processor Element

The arithmetic units in this discrete implementation are direct lookup tables implemented in static RAM. In a VLSI implementation these arithmetic units would be implemented with adders and small ROM lookup tables. Additional architectural

enhancements are made to PEs (1,1), and (1,2) to allow these two processors to operate as a very high throughput vector processor. The array architecture in vector mode is shown in Figure 6.3. The augmented processor is depicted in Figure 6.4.
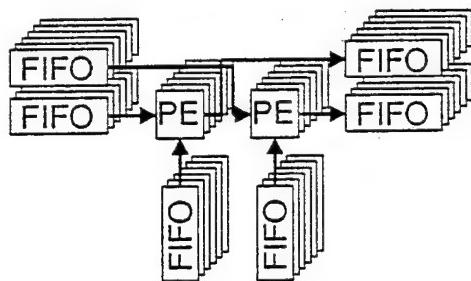
Figure 6.3: Block Diagram of Vector Mode Architecture
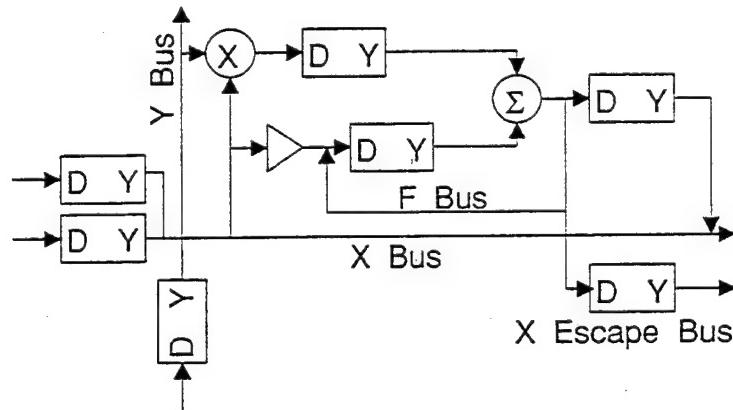
Figure 6.4: Augmented Processor Element

The X-bypass-bus of the enhanced PE is connected to the X-FIFOs, allowing two operands per cycle to be deposited on each of the enhanced processors. The X-escape bus of PE (1,1) allows the results to be flushed out of the processors in one clock cycle. The vector enhancement allows the Gauss machine to perform level 1 and level 2 operations very efficiently, and while the enhancement does not allow an addition of two operands to be performed directly, it may be performed in two cycles using

the accumulator. The vector processor can also perform pointwise multiplication of two vectors using a single clock cycle per operand pair.

6.2.1   Processor Control Signals

This section lists the processor control signals and their function. The control signals are registered on the processor boards. The signals are listed in Table 6.1.

The signals in Table 6.1 may be broken into several groups. These groups are:

- Address information: BA2-0, and PA2-0.

- FIFO Control: XIW*, YIW*, XOW*, XIR*, YIR*, XOR*, XIFLRT*, YI-FLRT*, and XOFLRT*.

- Adder RAM Control: ROE*, ARWE*.

- Multiplier RAM Control: MROE*, MRWE*.

- X-Bus Control: XBOE*, XBEN*, XFEN*, AREN*, and AROE*.

- Y-Bus Control: YBEN*.

- Processor Structure Control: PREN*, and SREN*.

- Processor Configuration Control: VECTORMODE and ARITHMODE.

- Miscellaneous: CLR*, RESBWE*, and RESBRE*.

6.3   Controller Implementation

The Gauss machine uses a microprogrammable controller. The heart of the controller is a single chip microsequencer with EPROM based microprogram store, the AMD Am29CPL154. The microcode store has a total of 512 words of microinstruction

storage. The microsequencer uses PLDs to decode its instructions for the array. The architecture of the controller is depicted in Figure 6.5. The Gauss machine controller has a pipeline delay model depicted in Figure 6.6.
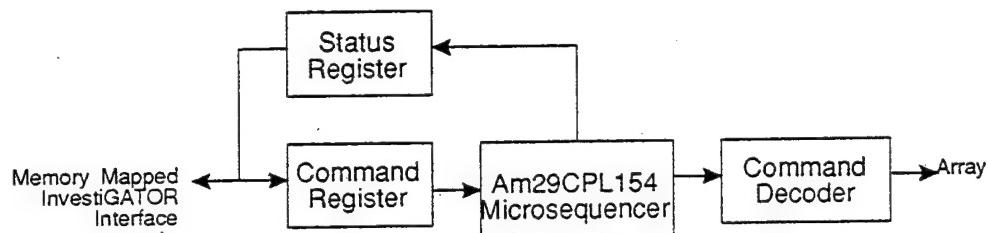
Figure 6.5: Block Diagram of Gauss Machine Controller Architecture
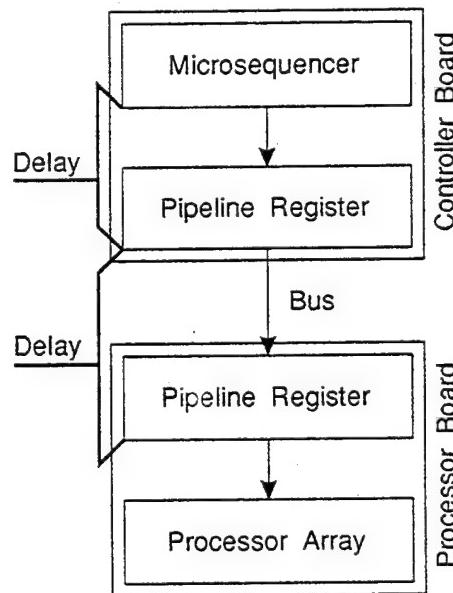
Figure 6.6: Gauss Machine Pipeline Delay Model

In order to perform an operation on the array, the InvestiGATOR will load some data into the array input FIFOs, and order the controller to perform the expected operation by writing a command to the command register. The InvestiGATOR then monitors the status register in order to determine when the computation is complete. Then the InvestiGATOR retrieves the results from the array output FIFOs.

This same method is used for programming the array multipliers and adders, except that there is no need to read back any results.

The chosen microsequencer, the Am29CPL154 has a relatively narrow output word (eight bits), yet the array has a substantial number of control lines as evidenced by Table 6.1. Fortunately, this does not present a problem because there are only a limited number of useful combinations of control signals. Therefore, the output word of the microsequencer is used as a command code or instruction and is decoded into the appropriate set of signals by the command decoder, see Figure 6.5.

6.4   Array Initialization

In order to perform useful operations on the array, the arithmetic elements must be initialized. There exist enhancements which are not visible in the block diagram of Figure 6.2 to allow programming of the multiplier. The adder can be programmed without any architectural enhancements.

The multiplier and architecture related to its programming is depicted in Figure 6.7. Control signals are indicated in the block diagram. The multiplier memory is addressed by the X-bus and Y-bus, and by the ARITHMODE signal. The multiplier data is loaded from the X-bus to the multiplier memory. Register output enable signals are indicated by an OE suffix while latch enable signals are indicated by an EN suffix. The write strobe for the memory is indicated by the MRWE* signal. The MRWE* signal is broadcast to all processors in the system so all of the multipliers must be programmed at the same time.

Programming of the multiplier proceeds as follows. The X- and Y- FIFOs are loaded by the InvestiGATOR. The InvestiGATOR sends a command to the Gauss machine controller to program a block of the multiplier memory. X- input FIFO
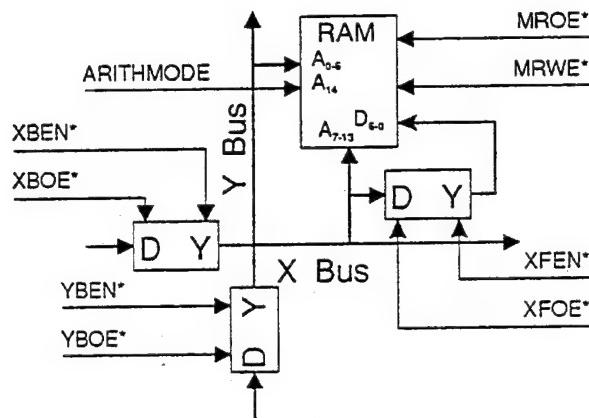
Figure 6.7: Processor Multiplier Programming Model

transmits the contents of the memory location across the X- bus to a register which outputs to the multiplier memory's data bus. Next, the address of the data word to be programmed is propagated across the array from the X- and Y- input FIFOs and the multiplier memory's write line is strobed. The process is repeated until the multiplier is programmed.

The adder and architecture related to its programming is depicted in Figure 6.8. The adder is programmed as follows. The adder data and addresses are loaded into the X- input FIFO. The least significant portion of the address is transmitted via the X-bus to the product output registers, controlled by PROE* and PREN*, with MROE* negated. Next, the most significant word of address is transmitted and loaded into the accumulator register, controlled by SROE* and SREN*. Finally, The actual data word is transmitted via the X-bus to the F-bus by way of the buffer controlled by XPOE*, and to the adder memory's data port. The adder memory write signal, ARWE*, is strobed, loading the data into the adder memory. This process is repeated until the adder is programmed.
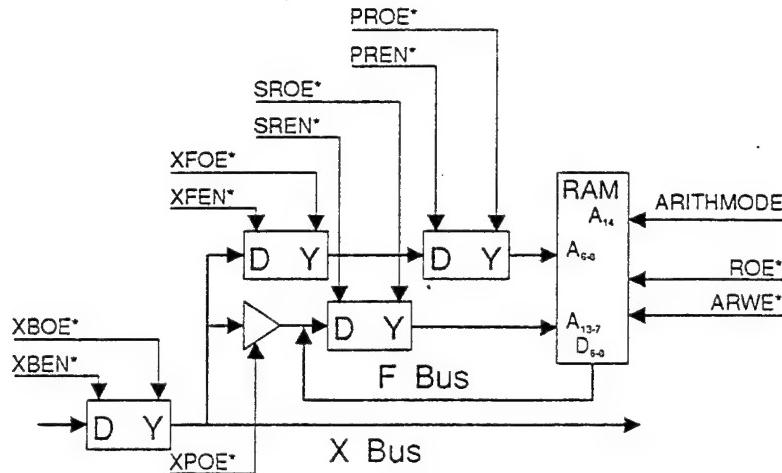
Figure 6.8: Processor Adder Programming Model

6.5 Conversion Engine Architecture

The forward conversion engine performs the task of generating the residues of the value input to the engine. This forward conversion is a relatively straightforward process once it is seen that the process may be accomplished simply by breaking the input values into a set of partial sums where each sum represents a range of bits of that number; in other words, suppose we wish to compute the residue modulo p of an L bit number N. We would note that the following congruence holds:

$$< N >_p \equiv \left(\sum_{i=0}^{L-1} a_i 2^i \right) \pmod{p},$$

where $a_i \in \{0, 1\}$ and are digits of the binary representation of N. Now, suppose $0 < J < K < L - 1$. Then

$$< N >_p \equiv \left[\sum_{i=0}^{J-1} a_i 2^i + \sum_{i=J}^{K-1} a_i 2^i + \sum_{i=K}^{L-1} a_i 2^i \right] \pmod{p}.$$

A   (mod p) operation may be added after each partial sum without changing the result:

$$< N >_p \equiv \left[\left(\sum_{i=0}^{J-1} a_i 2^i \right) (\mathrm{mod}\,p) + \left(\sum_{i=J}^{K-1} a_i 2^i \right) (\mathrm{mod}\,p) + \left(\sum_{i=K}^{L-1} a_i 2^i \right) (\mathrm{mod}\,p) \right] (\mathrm{mod}\,p).$$

This suggests that each partial sum, modulo p, can be computed using a small table, and the partial sums added together to form a sum which must be corrected modulo p. This is illustrated in Figure 1.2. In Figure 6.9a, conversion of a twenty-four bit input using two tables of order 2^{12} to produce an eight bit output is demonstrated. In Figure 6.9b, the same conversion is accomplished using three tables of order 2^8.
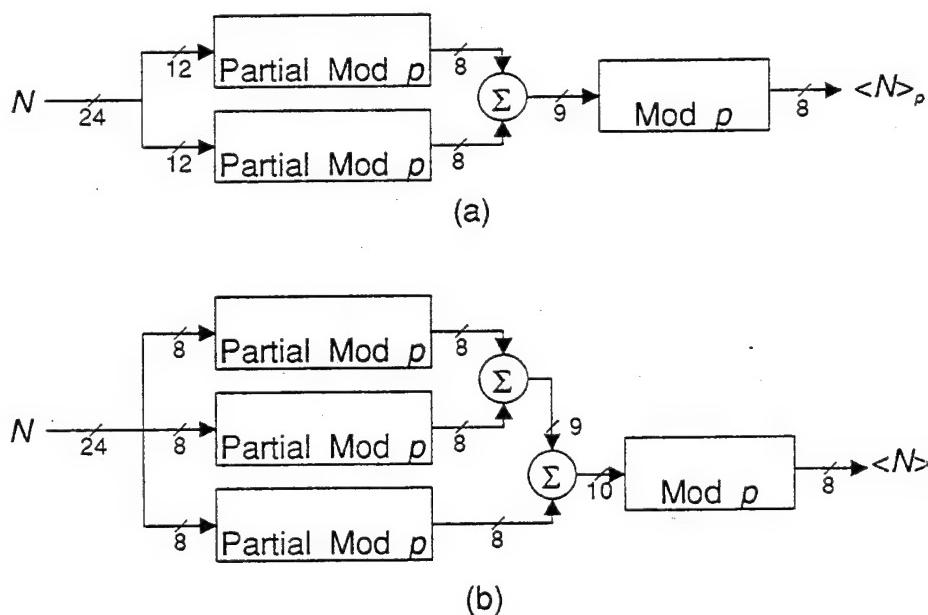
Figure 6.9: Forward Conversion Architecture

The forward conversion engine was not implemented in hardware since it would be relatively expensive to produce a discrete implementation. Instead, the forward conversion engine was implemented with a software architecture inspired by the above discussion. This was motivated by the low speed of a direct implementation of the

forward conversion using the standard sequence of divide, multiply, and subtract operations. In particular, the multiplication and division operations are particularly time consuming on the MC68030 (and most microprocessors). The source code in Section D.5 of Appendix D implements a high speed forward conversion based upon table lookup using small tables and minimal arithmetic (addition and subtraction only).

Similarly, the CRT engine hardware was too expensive to implement; emulation of the CRT was substituted. As for the forward conversion, the QRNS to CRNS to Gaussian integer conversion was implemented using a fast, table lookup based algorithm based upon the discussion in Section 1.3. The source code for this high performance implementation is included in Section D.5 of Appendix D.

6.6 Application Programmer's Interface

6.6.1 Overview

The system software for the Gauss Machine is divided into two parts: firmware for the backplane and the Application Programmer's Interface (API). This chapter describes the API which contains routines for linear algebra and communication between the host and the Gauss Machine. The API is written in THINK C 5.0 for the Macintosh.

The Application Programmer's Interface (API) contains roughly X subroutines that facilitates programming of the Gauss Machine. The idea behind the API is to provide fast prototyping environment for developing and testing new algorithms for the Gauss Machine. Therefore, the routines are not necessarily optimized for speed.

The API can be divided into "high-level" and "low-level" calls. The high-level routines often mimic Matlab statements, e.g., matrix-matrix, matrix-vector, vector-vector multiplication is handled by one routine called mult(). The low-level calls

implements the primitive operations from which the high-level routines are composed of, such as, memory management and communication between the host and the Gauss Machine. Furthermore, the algebra routines comes in two versions, one using floating-point arithmetic and the other using integer arithmetic.

Typically, the development of an algorithm for the Gauss Machine consists of the following steps:

- Program and test algorithm in Matlab.

- Port Matlab code into API calls.

- Test API code with the Gauss Machine.

- If optimization is of interest, rewrite code using the low-level API.

A complete listing of the API calls are found in Appendix X.

6.6.2   High-Level API Routines

Prototyping and testing signal processing/linear algebra algorithms are easily done in interactive packages like Matlab, Mathematica, Maple and Monarch/Siglab. The design of the high-level API was done with this in mind. The API routines imitates Matlab function calls which makes it easy to port an m-file or a Matlab script to a C program running on the Gauss Machine. The Matlab statements are simply exchanged to the corresponding API calls and, with some glue code, the port is complete.

The software was written in THINK C version 5.0 with the following libraries: ANSI, MacTraps. The code was compiled and run on a Mac IIx, 4Mb RAM, 4Mb virtual memory, System 7.0.

64

These are the THINK C settings under Edit, Options...

- **Language Settings**

  - ANSI Conformance

  - Check pointer types.

  - Language Extensions

  - THINK C

  - Strict Prototype Enforcements

  - Infer Prototypes

- **Compiler Settings**

  - Generate 68020 instructions

  - Generate 68881 instructions

  - Classes are indirect by default

  - Methods are virtual by default

  - Optimize monomorphic methods

  - \bslash p is unsigned char[]

- **Code Optimization**

  - Defer & combine stack adjust

  - Suppress redundant loads

  - Automatic Register Assignment Debugging

— Use source debugger

— Use second screen

— Always save session

These are the THINK C settings under `Project, Set Project Type...`

— Application

— File type APPL

— Partion (K) 384

— Size Flags 0000

The software consists of 5 "library" files (with corresponding header files) and one global header file:

`types.h`: Global type definitions.

`list.c`: Memory management routines. This software was originally written by R. F. Starr, 2639 Valley Field Dr., SugarLand, TX 77479 and was published in Dr. Dobbs Journal. `list.c` have been slightly modified.

`utils.c`: Utilities.

`conv.c`: Floating-point to fixed-point conversion routines.

`matrix.c`: Floating-point matrix algebra routines, memory management.

`int_matrix.c`: Integer matrix algebra routines, memory management.

The routines in `int_matrix.c` are identical to the routines in `matrix.c` except for those operations that are not defined for integers, i.e., division.

In order to compile these files as parts of a code resource, change all calls of `malloc()` to `NewPtr()` and `free()` to `DisposPrt()`. Furthermore, comment out the stdio routines, i.e., `printf()` and friends, in `utils.c`. It may be also necessary to change the ANSI library to the required library for code resources.

Note: Whenever the comments in the code and this document disagree, rely on this document.

6.6.3   Macros and Constants

file: `int_matrix.c`

```
#define COMP 0x4 /* marks compatible dimensions */

#define SCAL 0x8 /* marks one operand as a scalar */

#define INT(a) ((int)(a)) /* casts a to integer */

#define EQDIM(a, b) ( (a->rows == b->rows) && (a->cols ==
b->cols) ) /* checks if a and b has the same dimensions */
```

file: `matrix.c`

```
#define OOPS printf(oops: %d\n, __LINE__); /* debugging macro */

#define COMP 0x4 /* marks compatible dimesions */

#define SCAL 0x8 /* marks one operand as a scalar */

#define INT(a) ((int)(a)) /* casts a to integer */
```

file: `conv.h`

```
#define max(a, b) (a > b) ?  a :  b /* maximum of a and b */

#define min(a, b) (a < b) ?  a :  b /* minimum of a and b */
```

file: int_matrix.h]

```
#define deref(type,x) *((type*)(x)) /* not really useful */
```

file: matrix.h

```
#define SIZE(a) ((a)->rows * (a)->cols) /* computes number of
elements in matrix */

#define EQDIM(a, b) ( (a->rows == b->rows) && (a->cols ==
b->cols) ) /* checks if a and b has the same dimensions */

#define cmul(a, b, c, d, e, f);  a = (c) * (e) - (d) * (f); b =
(c) * (f) + (d) * (e);  /* complex multiply */

#define cabs(a, b) sqrt(((a) * (a) + (b) * (b))) /* compute
complex absolut value */
```

file: types.h

```
#define INTTYPE long /* integer data type */

#define FLOATTYPE double /* floating-point data type */

#define NOERR 0 /* OK return code */

#define CMPLX 0x1 /* marks a complex value */

#define REAL 0x2 /* marks a real value */
```

file: utils_h

```
#define PLAIN 0x0 /* Plain format */

#define MATLAB 0x1 /* Print in MATLAB style (with [ ] and ;) */
```

6.6.4 Function Descriptions.

matrix *add(matrix *a, matrix *b)

description: Adds matrices a and b.

arguments: `matrix *a, *b`   Input matrices.

returns: `matrix *`   The sum of a and b, NULL if error.

usage: `sum = add(a, b); /* sum = a + b`

matlab equivalent:

`>> sum = a + b; */`

file: `matrix.c`

matrix *appendcols(matrix *a, matrix *b)

description: Returns a matrix with b's columns appended to a ([a, b]). Naturally, a and b must have the same number of rows.

arguments: `matrix *a, *b`   Input matrices.

returns: `matrix *`   [a, b], NULL if error.

usage: `c = appendcols(a, b); /* c = [a, b]`

matlab equivalent:

```
>> c = [a, b]; */
```

file: `matrix.c`

matrix *appendrows(matrix *a, matrix *b)

description: Returns a matrix with b's rows appended to a ([a; b]). Naturally, a and b must have the same number of columns.

arguments: `matrix *a, *b`   Input matrices.

returns: `matrix *`   [a; b], NULL if error.

usage: `c = appendrows(a, b); /* c = [a; b]`

matlab equivalent:

```
>> c = [a; b]; */
```

file: `matrix.c`

matrix *assign(matrix *target, matrix *rows, matrix *cols, matrix* *source)

description: Puts the matrix source into a sub matrix of target indicated by rows and cols. That is, rows and cols defines a sub matrix of target (exactly like sub_matrix()) and this sub matrix is overwritten with data from the source matrix. This is analogous to the matlab statement target(rows, cols) = source. Needless to say, the sub matrix of target and source must be of the same dimensions. For example, suppose

target = [1 2 3 4; 5 6 7 8; 9 10 11 12],

source = [13 14; 15 16],

rows = [3 2] and cols = [1 2], the resulting matrix would be

[1 2 3 4; 15 16 7 8; 13 14 11 12].

If rows or cols is NULL, this means all the rows and all the columns of target. That is, target(rows,:) = source would be coded as assign(target, rows, NULL, source), and similarly, target(rows,:) = source would be coded as assign(target, rows, NULL, source).

arguments: `matrix *target`   Matrix to be written to.

`matrix *rows`   Row indexing matrix.

`matrix *cols`   Column indexing matrix

`matrix *source`   Matrix whose data will be written to target.

returns: `matrix *`   Copy of target with parts overwritten by source, NULL if error.

usage: `assign(target, rows, cols, source); // target(rows, cols) =`
`source`

`assign(target, rows, NULL, source); // target(rows, :)  = source`

`assign(target, NULL, cols, source); // target(:, cols) = source`

see also: `sub_matrix() temp_copy()`

note: Does not handle the case target(:,:) = source. For this case copy source with target = temp_copy(source).

file: `matrix.c`

void clear_error(void)

description: Clears the error string. This is typically done at start up or after recovering from an error.

arguments: nothing

returns: nothing

usage: `clear_error(); // Clear error messages`

see also: `get_error(), error(), print_error()`

file: `utils.c`

void close_GM(void)

description: Frees memory allocated to temporary matrices and cleans up. close_GM() should only be called once and be matched with a open_GM() call. To only free memory allocated by temporary matrices use kill_temp_list().

arguments: none

returns: nothing

usage: `close_GM(); // Clean up`

see also: `open_GM(), kill_temp_list()`

file: `utils.c`

int cmplx_promote(matrix *a, matrix *b)

description: Promotes, if necessary, the operands a and b to complex matrices, that is, if either a or b is complex then both a and b are converted to complex matrices.

arguments: matrix *a, *b   Matrices to be promoted.

returns: int   NOERR if successful, −1 if malloc failed

usage: res = cmplx_promote(a, b); /* Complex promotes a and b, OK if res == NOERR. */

see also: op_check()

file: matrix.c

matrix *conj(matrix *mat)

description: Returns a matrix equal to the complex conjugate of the input matrix.

arguments: matrix *mat   Input matrix.

returns: matrix *   The complex conjugated input matrix

usage: conj_A = conj(A); /* conj_A = -A

matlab equivalent:

>> conj_A = conj(A); */

file: matrix.c

matrix *copy_matrix(matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with new_matrix().
Note that it is the user's responsibility to free this matrix. Use copy_temp() to
get a temporary matrix (which will be freed by kill_temp_list() or close_GM()).

arguments: `matrix *source`   Matrix to be copied.

returns: `matrix *`   Copy of source. NULL if out of memory.

usage: `new = copy_matrix(old); // copy the matrix old to the matrix new`

see also: `kill_matrix(), new_temp(), copy_temp(), new_matrix(),`

`kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with
copy_matrix (with kill_matrix).

file: `matrix.c`

matrix *copy_temp(matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with new_temp()
and therefore, is a temporary matrix. To free ALL temporary matrices, use
kill_temp_list() or close_GM().

arguments: `matrix *source`   Matrix to be copied.

returns: `matrix *`   Copy of source. NULL if out of memory.

usage: new = copy_temp(old); // copy the matrix old to the temporary
matrix new

see also: kill_matrix(), copy_matrix(), new_temp(), new_matrix(),

kill_temp_list()

file: matrix.c

void error(char *msg)

description: Copies the string msg to the global error string. This routine is used to report
errors. The error string can be recovered by get_error(). Maximum length of
msg is 255 characters.

arguments: char *msg;   Error message to be copied to the global error string.

returns: nothing

usage: error(Division by zero is a bad idea); /* Division by zero error
message */

see also: get_error(), clear_error(), print_error()

file: utils.c

FLOATTYPE fixed2float(INTTYPE i)

description: Converts a fixed-point number to floating-point using the word length and
number of fractional bits set by init_conv().

arguments: INTTYPE i   Fixed-point number to be converted to floating-point.

returns: FLOATTYPE   Floating-point number representing the input argument.

usage: FLOATTYPE result;

INTTYPE int_result;

result = float2fixed(int_result); /* convert int_result to

floating-point */

see also: init_conv(), float2fixed(), mfloat2fixed(), mfixed2float()

file: conv.c

INTTYPE float2fixed(FLOATTYPE f)

description: Converts a floating-point number to fixed-point using the word length and
number of fractional bits set by init_conv().

arguments: FLOATTYPE f   Floating-point number to be converted to fixed-point.

returns: INTTYPE   Integer whose bits are the fixed-point representation of the input
argument.

usage: INTTYPE fixed_pi;

fixed_pi = float2fixed(3.141592654); // convert pi to fixed-point

see also: init_conv(), fixed2float(), mfloat2fixed(), mfixed2float()

file: conv.c

void get_error(char *error_string)

description: Copies the global error string to error_string. If an error has occurred the error string will contain an error message. error_string must be allocated to at least 255 characters by the caller.

arguments: char *error_string   Will contain a copy of the error message (if any).

returns: nothing

usage: `char err_str[255]; // make sure that err_str is at least 255 chars long`

`get_error(err_str); // get error message`

see also: error(), clear_error(), print_error()

file: utils.c

void GM2LV(matrix *a, TD1Hdl re, TD1Hdl im)

description: Copies data from GM matrix to Labview matrix. Note that re and im must be already allocated by the caller and of correct dimensions. If a is a real matrix then zeros will be put in im.

arguments: matrix *a   GM matrix whose data is to be copied to re and im.

TD1Hdl re, im   Handles to Labview matrix data structure.

returns: nothing

usage: `LV2GM(A, A_real, A_imag); // Copies data from A into A_real and`

`A_imag.`

see also: `LV2GM()`

file: `utils.c`

matrix *herm(matrix *mat)

description: Returns the conjugate transpose of the input matrix , that is, takes the hermitian of mat.

arguments: `matrix *mat`   Input matrix.

returns: `matrix *`   The conjugate transposed input matrix, NULL if error.

usage: `tran_A = herm(A); /* tran_A = A'`

`matlab equivalent:`

`>> A = A'; % Note ' not .', that is, conjugate transpose */`

see also: `transp()`

file: `matrix.c`

matrix *imag(matrix *mat)

description: Returns a matrix containing the imaginary part of mat.

arguments: `matrix *mat`   Input matrix.

returns: `matrix *`   Imaginary part of mat.

  usage: `im_part = imag(cmplx_matrix); /* im_part = Im[cmplx_matrix]`

    `matlab equivalent:`

    `>> im_part = imag(cmplx_matrix); */`

   file: `matrix.c`

`matrix *index_cols(matrix *mat, matrix *ind)`

description: Returns the columns of mat that are pointed out by ind. The elements of ind are truncated to integers (with mfloor()) and are used to pick out columns. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3.14 1.99], then the result would be a matrix of the form [3 2; 6 5; 9 8]. This is analogous to the matlab statement, mat(:, ind).

arguments: `matrix *mat`   Matrix to be indexed.

    `matrix *ind`   Column indexing matrix.

  returns: `matrix *`   The indexed input matrix, NULL if error.

   usage: `B = index_cols(mat, ind); /* B = mat(:, ind)`

    `matlab equivalent:`

    `>> B = mat(:, ind); */`

 see also: `index_rows()`, `index_rows_cols()`, `sub_matrix()`

   note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: matrix.c

matrix *index_rows(matrix *mat, matrix *ind)

description: Returns the rows of mat that are pointed out by ind. The elements of ind are truncated to integers (with mfloor()) and are used to pick out rows. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3.14 1.99], then the result would be a matrix of the form [7 8 9; 4 5 6]. This is analogous to the matlab statement, mat(ind, :).

arguments: matrix *mat   Matrix to be indexed.

matrix *ind   Row indexing matrix.

returns: matrix *   The indexed input matrix, NULL if error.

usage: B = index_rows(mat, ind); /* B = mat(ind, :)

matlab equivalent:

>> B = mat(ind, :)   */

see also: index_cols(), index_rows_cols(), sub_matrix()

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: matrix.c

matrix *index_rows_cols(matrix *mat, matrix *ind_a, matrix *ind_b)

description: Returns the rows and columns of mat that are pointed out by ind_a and ind_b. The elements of ind_a and ind_b are truncated to integers (with mfloor()) and are used to pick out rows and columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind_a = [3.14 1.99] and ind_b = [2.1], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(ind_a, ind_b).

arguments: `matrix *mat`   Matrix to be indexed.

`matrix *ind_a`   Row indexing matrix.

`matrix *ind_b`   Column indexing matrix.

returns: `matrix *`   The indexed input matrix, NULL if error.

usage: `B = index_rows_cols(mat, ind_a, ind_b); /* B = mat(ind_a, ind_b)`

`matlab equivalent:`

`>> B = mat(ind_a, ind_b); */`

see also: `index_rows()`, `index_cols()`, `sub_matrix()`

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: `matrix.c`

void init_GM(void)

description: Initializes everything. Clears the global error string (see error()), sets output print format to MATLAB and 8 digits (see init_print()) and initializes

memory management (see init_temp_list()). init_GM() should only be called once and be matched with a close_GM() call.

arguments: none

returns: nothing

usage: init_GM(); // Initialize everything

see also: close_GM()

file: utils.c

void init_conv(int wlen, int fbits)

description: Initializes the conversion routines and sets the word length and number of fractional bits for the fixed-point representation. The fixed-point numbers are assumed to be signed. Thus, for a word length of 8 and 3 fractional bits, one bit would be the sign bit and 4 bits will be left for the integer part. This means that number between 1111.111 and -1111.111 (±15.875) can be represented. Maximal word length is 32.

arguments: int wlen   Number of bits per word

int fbits   Number of fractional bits

returns: nothing

usage: init_conv(21, 6); /* Use 21 bit signed words; 6 fractional bits, 14 integer bits */

see also: `float2fixed()`, `fixed2float()`, `mfloat2fixed()`, `mfixed2float()`

file: `conv.c`

void init_print(int sdig, int format)

description: Set number of significant digits and format used by printm() and int_printm(). The format can either be PLAIN or MATLAB. MATLAB gives a text output that is easily imported into matlab. PLAIN on the other hand is a bit easier read.

arguments: `int sdig`   Number of significant digits.

`int format`   Output format style, either MATLAB or PLAIN.

returns: nothing

usage: `init_print(6, MATLAB); // 6 significant digits and MATLAB output format`

see also: `printm()`, `int_printm()`, `init_GM()`

file: `utils.c`

int init_temp_list(void)

description: Initialize the list for allocation of temporary matrices

arguments: none

returns: `int`   NOERR if all right, −1 if out of memory

usage: `err = init_temp_list(); /* Initialize temp matrices, inspect err`

   `for errors.  */`

see also: `init_GM()`

note: init_temp_list() is normally called from init_GM();

file: `matrix.c`

`int_matrix *int_add(int_matrix *a, int_matrix *b)`

description: Adds matrices a and b.

arguments: `int_matrix *a, *b`   Input matrices.

returns: `int_matrix *`   The sum of a and b, NULL if error.

usage: `sum = int_add(a, b); /* sum = a + b`

   `matlab equivalent:`

   `>> sum = a + b; */`

file: `int_matrix.c`

`int_matrix *int_appendcols(int_matrix *a, int_matrix *b)`

description: Returns a matrix with b's columns appended to a ([a, b]). Naturally, a and b
must have the same number of rows.

arguments: `int_matrix *a, *b`   Input matrices.

returns: `int_matrix *`   [a, b], NULL if error.

usage: `c = int_appendcols(a, b); /* c = [a, b]`

```
matlab equivalent:

>> c = [a, b]; */
```

file: `int_matrix.c`

int_matrix *int_appendrows(int_matrix *a, int_matrix *b)

description: Returns a matrix with b's rows appended to a ([a; b]). Naturally, a and b must have the same number of columns.

arguments: `int_matrix *a, *b`   Input matrices.

returns: `int_matrix *`   [a; b], NULL if error.

usage: `c = int_appendrows(a, b); /* c = [a; b]`

```
matlab equivalent:

>> c = [a; b]; */
```

file: `int_matrix.c`

int_matrix *int_assign(int_matrix *target, int_matrix *rows, int_matrix *cols, matrix *source)

description: Puts the matrix source into a sub matrix of target indicated by rows and cols. That is, rows and cols defines a sub matrix of target (exactly like int_sub_matrix()) and this sub matrix is overwritten with data from the source

matrix. This is analogous to the matlab statement target(rows, cols) = source. Needless to say, the sub matrix of target and source must be of the same dimensions. For example, suppose

target = [1 2 3 4; 5 6 7 8; 9 10 11 12],

source = [13 14; 15 16],

rows = [3 2] and cols = [1 2], the resulting matrix would be

[1 2 3 4; 15 16 7 8; 13 14 11 12].

If rows or cols is NULL, this means all the rows and all the columns of target. That is, target(rows,:) = source would be coded as assign(target, rows, NULL, source), and similarly, target(rows,:) = source would be coded as int_assign(target, rows, NULL, source).

arguments: `int_matrix *target`   Matrix to be written to.

`int_matrix *rows`   Row indexing matrix.

`int_matrix *cols`   Column indexing matrix

`int_matrix *source`   Matrix whose data will be written to target.

returns: `int_matrix *`   Copy of target with parts overwritten by source, NULL if error.

usage: `int_assign(target, rows, cols, source); // target(rows, cols) = source`

`int_assign(target, rows, NULL, source); // target(rows, :) = source`

```
int_assign(target, NULL, cols, source); // target(:, cols) =
source
```

see also: `int_sub_matrix()`, `int_temp_copy()`

note: Does not handle the case target(:,:) = source. For this case copy source with
target = int_temp_copy(source).

file: `int_matrix.c`

int int_cmplx_promote(int_matrix *a, int_matrix *b)

description: Promotes, if necessary, the operands a and b to complex matrices, that is, if
either a or b is complex then both a and b are converted to complex matrices.

arguments: `int_matrix *a, *b`   Matrices to be promoted.

returns: `int`   NOERR if successful, −1 if malloc failed

usage: `res = int_cmplx_promote(a, b); /* Complex promotes a and b, OK if
res == NOERR. */`

see also: `int_op_check()`

file: `int_matrix.c`

int_matrix *int_conj(int_matrix *mat)

description: Returns a matrix equal to the complex conjugate of the input matrix.

arguments: `int_matrix *mat`   Input matrix.

returns: int_matrix *   The complex conjugated input matrix

usage: conj_A = int_conj(A); /* conj_A = -A

matlab equivalent:

>> conj_A = conj(A); */

file: int_matrix.c

int_matrix *int_copy_matrix(int_matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with

int_new_matrix(). Note that it is the user's responsibility to free this matrix.

Use int_copy_temp() to get a temporary matrix (which will be freed by

kill_temp_list() or close_GM()).

arguments: int_matrix *source   Matrix to be copied.

returns: int_matrix *   Copy of source. NULL if out of memory.

usage: new = int_copy_matrix(old); // copy the matrix old to the matrix

new

see also: int_kill_matrix(), int_new_temp(), int_copy_temp(),

int_new_matrix(),

kill_temp_list()

note: It is the user's responsibility to free any matrix that has been allocated with

int_copy_matrix (with int_kill_matrix).

file: `int_matrix.c`

int_matrix *int_copy_temp(int_matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with

int_new_temp() and therefore, is a temporary matrix. To free ALL temporary

matrices, use kill_temp_list() or close_GM().

arguments: `int_matrix *source`   Matrix to be copied.

returns: `int_matrix *`   Copy of source. NULL if out of memory.

usage: `new = int_copy_temp(old); // copy the matrix old to the temporary`

`matrix new`

see also: `int_kill_matrix()`, `int_copy_matrix()`, `int_new_temp()`,

`int_new_matrix()`,

`kill_temp_list()`

file: `int_matrix.c`

int_matrix *int_herm(int_matrix *mat)

description: Returns the conjugate transpose of the input matrix, that is, takes the

hermitian of mat.

arguments: `int_matrix *mat`   Input matrix.

returns: `int_matrix *`   The conjugate transposed input matrix, NULL if error.

usage: `tran_A = int_herm(A); /* tran_A = A'`

matlab equivalent:

`>> A = A'; % Note ' not .', that is, conjugate transpose */`

see also: `int_transp()`

file: `int_matrix.c`

`int_matrix *int_imag(int_matrix *mat)`

description: Returns a matrix containing the imaginary part of mat.

arguments: `int_matrix *mat`   Input matrix.

returns: `int_matrix *`   Imaginary part of mat.

usage: `im_part = int_imag(cmplx_matrix); /* im_part = Im[cmplx_matrix]`

matlab equivalent:

`>> im_part = imag(cmplx_matrix); */`

file: `int_matrix.c`

`int_matrix *int_index_cols(int_matrix *mat, int_matrix *ind)`

description: Returns the columns of mat that are pointed out by ind. The elements of ind are used to pick out columns. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3 1], then the result would be a matrix of the form [3 2; 6 5; 9 8]. This is analogous to the matlab statement, mat(:, ind).

arguments: `int_matrix *mat`   Matrix to be indexed.

`int_matrix *ind`   Column indexing matrix.

returns: `int_matrix *`   The indexed input matrix, NULL if error.

usage: `B = int_index_cols(mat, ind); /* B = mat(:, ind)`

`matlab equivalent:`

`>> B = mat(:, ind); */`

see also: `int_index_rows()`, `int_index_rows_cols()`, `int_sub_matrix()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

`int_matrix *int_index_rows(int_matrix *mat, int_matrix *ind)`

description: Returns the rows of mat that are pointed out by ind. The elements of ind are used to pick out rows. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3 1], then the result would be a matrix of the form [7 8 9; 4 5 6]. This is analogous to the matlab statement, mat(ind, :).

arguments: `int_matrix *mat`   Matrix to be indexed.

`int_matrix *ind`   Row indexing matrix.

returns: `int_matrix *`   The indexed input matrix, NULL if error.

usage: B = int_index_rows(mat, ind); /* B = mat(ind, :)

matlab equivalent:

>> B = mat(ind, :)   */

see also: int_index_cols(), int_index_rows_cols(), int_sub_matrix()

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: int_matrix.c

int_index_rows_cols(int_matrix *mat, int_matrix *ind_a, int_matrix* ind_b)

description: Returns the rows and columns of mat that are pointed out by ind_a and ind_b.
The elements of ind_a and ind_b are used to pick out rows and columns,
respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind_a = [3 1]
and ind_b = [2], then the result would be a matrix of the form [8; 2] (that is,
elements (3,2) and (1,2)). This is analogous to the matlab statement,
mat(ind_a, ind_b).

arguments: int_matrix *mat   Matrix to be indexed.

int_matrix *ind_a   Row indexing matrix.

int_matrix *ind_b   Column indexing matrix.

returns: int_matrix *   The indexed input matrix, NULL if error.

usage: B = int_index_rows_cols(mat, ind_a, ind_b); /* B = mat(ind_a,
ind_b)

```
        matlab equivalent:

        >> B = mat(ind_a, ind_b); */
```

see also: `int_index_rows()`, `int_index_cols()`, `int_sub_matrix()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

void int_kill_matrix(int_matrix *mat)

description: Frees memory used by mat. Note that mat should have been allocated with int_new_matrix() or int_copy_matrix(). If mat is a temporary matrix, that is, allocated explicitly or implicitly with int_new_temp() or int_copy_temp(), then kill_temp_list() should be used.

arguments: `int_matrix *mat`   Matrix to be freed, must have been allocated by int_new_matrix() or int_copy_matrix()

returns: nothing

usage: `int_kill_matrix(A); // Free memory allocated for A`

see also: `int_copy_matrix()`, `int_new_temp()`, `int_copy_temp()`,
`int_new_matrix()`,

`kill_temp_list()`

note: The matrix mat must have been allocated by int_new_matrix() or int_copy_matrix()

file: int_matrix.c

int int_max_index(int_matrix *mat)

description: Return index to maximal element of mat.

arguments: int_matrix *mat   Input matrix.

returns: int   Index to the maximal element of mat.

usage: max_i = int_max_index(mat); /* max(mat) = mat[max_i]

matlab equivalent:

>> max_i = find(mat == max(mat(:))); */

file: int_matrix.c

int int_min_index(int_matrix *mat)

description: Return index to minimal element of mat.

arguments: int_matrix *mat   Input matrix.

returns: int   Index to the minimal element of mat.

usage: min_i = int_min_index(mat); /* min(mat) = mat[min_i]

matlab equivalent:

>> min_i = find(mat == min(mat(:))); */

file: int_matrix.c

int_matrix *int_minus(int_matrix *mat)

description: Returns a matrix equal to the negative of input matrix.

arguments: int_matrix *mat   Input matrix.

returns: int_matrix *   The negated input matrix.

usage: minus_A = int_minus(A); /* minus_A = -A

      matlab equivalent:

      >> minus_A = -A; */

file: int_matrix.c

int_matrix *int_mul(int_matrix *a, int_matrix *b)

description: Multiplication of matrices a and b.

arguments: int_matrix *a, *b   Input matrices.

returns: int_matrix *   The product of a and b, NULL if error.

usage: prod = int_pmul(a, b); /* pprod = a * b

      matlab equivalent:

      >> prod = a * b; */

file: int_matrix.c

int int_mul_check(int_matrix *a, int_matrix *b)

description: Check field type and dimensions. First a and b are complex promoted by
int_cmplx_promote(). Returns COMP if a and b have the same dimensions.
Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b
are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR
ed to the returned value. Thus, if type = int_op_check(a,b), then (type &
CMPLX) will be true if either a or b is complex, (type & REAL) will be true
if both a and b are real, (type & COMP) will be true if dimensions match or if
a or b is a scalar, (type & SCAL) will be true if a or b is a scalar.

arguments: int_matrix *a, *b   matrices to be checked

returns: int   The bits are set according to the description above.

usage: type = int_mul_check(a, b); /* checks dimensions of a, b. Bits
in type will be set in type according to the description above */

see also: int_cmplx_promote(), int_op_check()

note: int_mul_check() is used by int_mul(). A scalar is compatible with any matrix.

file: int_matrix.c

int_matrix *int_new_matrix(int rows, int cols, int type)

description: Allocates memory for a matrix with dimension rows and cols and of type type
(CMPLX or REAL). Note that it is the user's responsibility to free this
matrix. Use int_new_temp() to allocate memory for a temporary matrix
(which will be freed by kill_temp_list() or close_GM()).

arguments: `int rows, cols`   Dimensions of matrix to be allocated

`int type`   Type of matrix, CMPLX for a complex matrix and REAL for a real matrix

returns: `int_matrix *`   Matrix of the requested size and type. NULL if out of memory.

usage: `mat = int_new_matrix(3, 5, CMPLX); /* Allocate memory for a 3x5 complex matrix */`

see also: `int_copy_matrix()`, `int_new_temp()`, `int_copy_temp()`, `int_kill_matrix()`,

`kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with int_new_matrix() (with int_kill_matrix()).

file: `int_matrix.c`

`int_matrix *int_new_temp(int rows, int cols, int type)`

description: Allocates memory for a temporary matrix with dimension rows and cols and of type. Note that it is the user's responsibility to free this matrix. To free ALL temporary matrices, use kill_temp_list() or close_GM().

arguments: `int rows, cols`   Dimensions of matrix to be allocated.

`int type`   Type of matrix, CMPLX for a complex matrix and REAL for a real matrix.

returns: `int_matrix *`   Matrix of the requested size and type. NULL if out of

memory.

usage: `mat = int_new_temp(3, 5, CMPLX); // Allocate memory for a`

`temporary 3x5 complex matrix`

see also: `int_kill_matrix()`, `int_copy_matrix()`, `int_copy_temp()`,

`int_new_matrix()`,

`kill_temp_list()`

file: `int_matrix.c`

int int_op_check(int_matrix *a, int_matrix *b)

description: Check field type and dimensions. First a and b are complex promoted by
int_cmplx_promote(). Returns COMP if a and b have the same dimensions.
Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b
are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR
ed to the returned value. Thus, if type = int_op_check(a,b), then (type &
CMPLX) will be true if either a or b is complex, (type & REAL) will be true
if both a and b are real, (type & COMP) will be true if dimensions match,
(type & SCAL) will be true if either a or b is a scalar.

arguments: `int_matrix *a, *b`   Matrices to be checked

returns: `int`   The bits are set according to the description above.

usage: `type = int_op_check(a, b); /* checks dimensions of a, b.  Bits in`

`type will be set in type according to the description above */`

98

see also: `int_cmplx_promote()`, `int_mul_check()`

note: int_op_check() is used by int_add(), int_pmul(), int_pdiv(). A scalar is compatible with any matrix.

file: `int_matrix.c`

int_matrix *int_pmul(int_matrix *a, int_matrix *b)

description: Point wise multiplication of matrices a and b.

arguments: `int_matrix *a, *b`   Input matrices.

returns: `int_matrix *`   The point wise product of a and b, NULL if error.

usage: `pprod = int_pmul(a, b); /* pprod = a .* b`

matlab equivalent:

`>> pprod = a .* b; */`

file: `int_matrix.c`

void int_printm(int_matrix *mat)

description: Prints an integer matrix to stdout using the number of significant digits and output style set by init_print().

arguments: `int_matrix *mat`   Matrix to be printed

returns: `void`

usage: `printm(Rxx); // prints Rxx to stdout`

see also: `init_print() printm()`

file: `utils.c`

`int_matrix *int_range(INTTYPE from, INTTYPE step, INTTYPE to)`

description: Creates a vector like matlab's from:step:to. If step is 0, then the step size is set to 1. For example, int_range(1, 2, 7) results in [1 3 5 7], int_range(3, 0, 5) results in [3 4 5].

arguments: `INTTYPE from`   Start value.

           `INTTYPE step`   Step size.

           `INTTYPE to`   Stop value.

returns: `int_matrix *`   Vector with elements starting at from and stopping at to, spaced by step. NULL if error.

usage: `int_range(from, step, to); // from:step:to`

       `int_range(from, 0, to); // from:to`

file: `int_matrix.c`

`int_matrix *int_real(int_matrix *mat)`

description: Returns a matrix containing the real part of mat.

arguments: `int_matrix *mat`   Input matrix.

returns: int_matrix *   Real part of mat.

usage: real_part = int_real(cmplx_matrix); /* real_part =
Re[cmplx_matrix]

matlab equivalent:

>> real_part = real(cmplx_matrix); */

file: int_matrix.c

int_matrix *int_scl2mat(INTTYPE re, INTTYPE im, int type);

description: Creates a 1x1 matrix from the scalar (re + j*im), if type is CMPLX. If type is
REAL the imaginary part is ignored.

arguments: INTTYPE re   Real part.

INTTYPE im   Imaginary part

returns: int_matrix *   1x1 matrix with the element (re + j*im). NULL if error.

usage: scalar_mat = int_scal2mat(3, 2, CMPLX); // scalar_mat(1,1) = 3 +
j*2

scalar_mat = int_scal2mat(3, 2, REAL); // scalar_mat(1,1) = 3

int_sub_matrix(int_matrix *mat, int_matrix *rows, int_matrix* int_matrix *cols);

description: Returns the rows and columns of mat that are pointed out by (the matrices)
rows and cols. The elements of rows and cols are used to pick out rows and
columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and rows =

[3 1] and cols = [2], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(rows, cols). To just index rows, like matlab's mat(rows, :), set cols to NULL. Similarly, to just index columns, like matlab's mat(:, cols), set rows to NULL. By using this scheme all indexing can be done with int_sub_matrix().

arguments: `int_matrix *mat`   Matrix to be indexed.

`int_matrix *rows`   Row indexing matrix.

`int_matrix *cols`   Column indexing matrix.

returns: `int_matrix *`   The indexed input matrix, NULL if error.

usage: `B = int_sub_matrix(mat, rows, cols); // B = mat(rows, cols)`

`B = int_sub_matrix(mat, rows, NULL); // B = mat(rows, :)`

`B = int_sub_matrix(mat, NULL, cols); /* B = mat(:, cols)`

`matlab equivalent:`

`>> B = mat(rows, cols); */`

see also: `int_index_rows()`, `int_index_cols()`, `int_index_rows_cols()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

`int_matrix *int_transp(int_matrix *mat)`

description: Returns the transpose of the input matrix. Note: does not conjugate elements. Use int_herm() for conjugate transpose (hermitian).

arguments: `int_matrix *mat`   Input matrix.

returns: `int_matrix *`   The transposed input matrix, NULL if error.

usage: `tran_A = int_transp(A); /* tran_A = A.'`

matlab equivalent:

`>> A = A.'; % Note .' not ', that is, does not conjugate */`

see also: `int_herm()`

file: `int_matrix.c`

void kill_matrix(matrix *mat)

description: Frees memory used by mat. Note that mat should have been allocated with new_matrix() or copy_matrix(). If mat is a temporary matrix, that is, allocated explicitly or implicitly with new_temp() or copy_temp(), then kill_temp_list() should be used.

arguments: `matrix *mat`   Matrix to be freed, must have been allocated by new_matrix() or copy_matrix()

returns: nothing

usage: `kill_matrix(A); // Free memory allocated for A`

see also: copy_matrix(), new_temp(), copy_temp(), new_matrix(),

kill_temp_list()

note: The matrix mat must have been allocated by new_matrix() or copy_matrix()

file: matrix.c

void kill_temp_list(void)

description: Frees memory allocated by ALL temporary matrices. The temporary matrices are allocated, explicitly or implicitly, with new_temp() or copy_temp.

arguments: none

returns: nothing

usage: kill_temp_list(); // Free memory allocated by all temporary matrices

see also: kill_matrix(), copy_matrix(), new_temp(), new_matrix(), copy_temp(), close_GM()

file: matrix.c

matrix *LV2GM(TD1Hdl re, TD1Hdl im)

description: Copies Labview matrices to Gauss Machine (GM) matrices. No explicit memory allocation is necessary. The memory management is handled internally.

arguments: `TD1Hdl re, im`   Handles to Labview matrix data structures.

returns: `matrix *`   Gauss machine matrix with data from the input arguments re and im.

usage: `A = LV2GM(A_real, A_imag); /* Create a GM matrix with real part data from A_real and imaginary data from A_imag.  */`

see also: `GM2LV()`

file: `utils.c`

int max_index(matrix *mat)

description: Return index to maximal element of mat.

arguments: `matrix *mat`   Input matrix.

returns: `int`   Index to the maximal element of mat.

usage: `max_i = max_index(mat); /* max(mat) = mat[max_i]`

`matlab equivalent:`

`>> max_i = find(mat == max(mat(:))); */`

file: `matrix.c`

matrix *mfixed2float(int_matrix *mat)

description: Converts a fixed-point matrix to floating-point using the word length and number of fractional bits set by init_conv().

arguments: `int_matrix *mat`   Fixed-point matrix to be converted to floating-point.

returns: `matrix *`   Floating-point matrix corresponding to the fixed-point matrix.

usage: `int_matrix *int_Rxx;`

`matrix *Rxx;`

`Rxx = mfixed2float(int_Rxx); // convert int_Rxx to floating-point`

see also: `init_conv(), fixed2float(), mfloat2fixed(), fixed2float()`

file: `conv.c`

int_matrix *mfloat2fixed(matrix *mat)

description: Converts a floating-point matrix to fixed-point using the word length and number of fractional bits set by init_conv().

arguments: `matrix *mat`   Floating-point matrix to be converted to fixed-point.

returns: `int_matrix *`   Integer matrix containing the fixed-point representation of the input floating-point matrix.

usage: `int_matrix *int_Rxx;`

`matrix *Rxx;`

`Rxx = mfixed2float(int_Rxx); // convert int_Rxx to floating-point`

see also: `init_conv(), fixed2float(), mfloat2fixed(), fixed2float()`

file: `conv.c`

matrix *mfloor(matrix *mat)

description: Returns a matrix with truncated elements of mat, that is, performs the floor function (rounding to closest smallest integer) on mat.

arguments: `matrix *mat`   Input matrix.

returns: `matrix *`   The "floored" input matrix, NULL if error.

usage: `int_A = mfloor(A); /* int_A = floor(A)`

```
matlab equivalent:

>> int_A = floor(A); */
```

note: The output is still a floating point matrix, even though the elements are truncated to integers.

file: `matrix.c`

int min_index(matrix *mat)

description: Return index to minimal element of mat.

arguments: `matrix *mat`   Input matrix.

returns: `int`   Index to the minimal element of mat.

usage: `min_i = min_index(mat); /* min(mat) = mat[min_i]`

```
matlab equivalent:

>> min_i = find(mat == min(mat(:))); */
```

file: `matrix.c`

matrix *minus(matrix *mat)

description: Returns a matrix equal to the negative of input matrix.

arguments: `matrix *mat`   Input matrix.

returns: `matrix *`   The negated input matrix.

usage: `minus_A = minus(A); /* minus_A = -A`

        `matlab equivalent:`

        `>> minus_A = -A; */`

file: `matrix.c`

matrix *mul(matrix *a, matrix *b)

description: Multiplication of matrices a and b.

arguments: `matrix *a, *b`   Input matrices.

returns: `matrix *`   The product of a and b, NULL if error.

usage: `prod = pmul(a, b); /* pprod = a * b`

        `matlab equivalent:`

        `>> prod = a * b; */`

file: `matrix.c`

<center>int mul_check(matrix *a, matrix *b)</center>

description: Check field type and dimensions. First a and b are complex promoted by cmplx_promote(). Returns COMP if a and b have the same dimensions. Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR ed to the returned value. Thus, if type = op_check(a,b), then (type & CMPLX) will be true if either a or b is complex, (type & REAL) will be true if both a and b are real, (type & COMP) will be true if dimensions match or if a or b is a scalar, (type & SCAL) will be true if a or b is a scalar.

arguments: matrix *a, *b   matrices to be checked

returns: int   The bits are set according to the description above.

usage: `type = mul_check(a, b); /* checks dimensions of a, b.  Bits in type will be set in type according to the description above */`

see also: cmplx_promote(), op_check()

note: mul_check() is used by mul(). A scalar is compatible with any matrix.

file: matrix.c

<center>matrix *new_matrix(int rows, int cols, int type)</center>

description: Allocates memory for a matrix with dimension rows and cols and of type (CMPLX or REAL). Note that it is the user's responsibility to free this matrix. Use new_temp() to allocate memory for a temporary matrix (which will be freed by kill_temp_list() or close_GM()).

arguments: `int rows, cols`   Dimensions of matrix to be allocated

`int type`   Type of matrix, CMPLX for a complex matrix and REAL for a
real matrix

returns: `matrix *`   Matrix of the requested size and type. NULL if out of memory.

usage: `mat = new_matrix(3, 5, CMPLX); /* Allocate memory for a 3x5
complex matrix */`

see also: `copy_matrix(), new_temp(), copy_temp(), kill_matrix(),
kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with
new_matrix() (with kill_matrix()).

file: `matrix.c`

matrix *new_temp(int rows, int cols, int type)

description: Allocates memory for a temporary matrix with dimension rows and cols and
of type. Note that it is the user's responsibility to free this matrix. To free
ALL temporary matrices, use kill_temp_list() or close_GM().

arguments: `int rows, cols`   Dimensions of matrix to be allocated.

`int type`   Type of matrix, CMPLX for a complex matrix and REAL for a
real matrix.

returns: `matrix *`   Matrix of the requested size and type. NULL if out of memory.

usage: `mat = new_temp(3, 5, CMPLX); // Allocate memory for a temporary`
3x5 complex matrix

see also: `kill_matrix()`, `copy_matrix()`, `copy_temp()`, `new_matrix()`,

`kill_temp_list()`

file: `matrix.c`

int op_check(matrix *a, matrix *b)

description: Check field type and dimensions. First a and b are complex promoted by cmplx_promote(). Returns COMP if a and b have the same dimensions. Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR ed to the returned value. Thus, if type = op_check(a,b), then (type & CMPLX) will be true if either a or b is complex, (type & REAL) will be true if both a and b are real, (type & COMP) will be true if dimensions match, (type & SCAL) will be true if either a or b is a scalar.

arguments: `matrix *a, *b`   Matrices to be checked

returns: `int`   The bits are set according to the description above.

usage: `type = op_check(a, b); /* checks dimensions of a, b.  Bits in`
`type will be set in type according to the description above */`

see also: `cmplx_promote()`, `mul_check()`

note: op_check() is used by add(), pmul(), pdiv(). A scalar is compatible with any matrix.

file: `matrix.c`

matrix *pdiv(matrix *a, matrix *b)

description: Point wise division of matrices a and b.

arguments: `matrix *a, *b`   Input matrices.

returns: `matrix *`   The point wise division of a and b, NULL if error.

usage: `pprod = pdiv(a, b); /* pprod = a ./ b`

matlab equivalent:

`>> pprod = a ./ b; */`

file: `matrix.c`

matrix *pmul(matrix *a, matrix *b)

description: Point wise multiplication of matrices a and b.

arguments: `matrix *a, *b`   Input matrices.

returns: `matrix *`   The point wise product of a and b, NULL if error.

usage: `pprod = pmul(a, b); /* pprod = a .* b`

matlab equivalent:

`>> pprod = a .* b; */`

file: matrix.c

void print_error(void)

description: Prints the global error string to stderr.

arguments: none

returns: nothing

usage: print_error(); // Print error string to stderr.

see also: get_error(), clear_error(), error()

file: utils.c

void printm(matrix *mat)

description: Prints a matrix to stdout using the number of significant digits and output
style set by init_print().

arguments: matrix *mat   Matrix to be printed

returns: void

usage: printm(Rxx); // prints Rxx to stdout

see also: init_print() int_printm()

file: utils.c

matrix *range(FLOATTYPE from, FLOATTYPE step, FLOATTYPE to)

description: Creates a vector like matlab's from:step:to. If step is 0, then the step size is
set to 1. For example, range(1, 2, 7) results in [1 3 5 7], range(3, 0, 5) results
in [3 4 5].

arguments: FLOATTYPE from   Start value.

           FLOATTYPE step   Step size.

           FLOATTYPE to   Stop value.

returns: matrix *   Vector with elements starting at from and stopping at to,
spaced by step. NULL if error.

usage: range(from, step, to); // from:step:to

       range(from, 0, to); // from:to

file: matrix.c

matrix *real(matrix *mat)

description: Returns a matrix containing the real part of mat.

arguments: matrix *mat   Input matrix.

returns: matrix *   Real part of mat.

usage: real_part = real(cmplx_matrix); /* real_part = Re[cmplx_matrix]

       matlab equivalent:

       >> real_part = real(cmplx_matrix); */

file: `matrix.c`

matrix *scl2mat(double re, double im, int type);

description: Creates a 1x1 matrix from the scalar (re + j*im), if type is CMPLX. If type is REAL the imaginary part is ignored.

arguments: `FLOATTYPE re`   Real part.

            `FLOATTYPE im`   Imaginary part

returns: `matrix *`   1x1 matrix with the element (re + j*im). NULL if error.

usage: `scalar_mat = scal2mat(3.14, 2.5, CMPLX); // scalar_mat(1,1) = 3.14 + j*2.5`

     `scalar_mat = scal2mat(3.14, 2.5, REAL); // scalar_mat(1,1) = 3.14`

void set_fname(char *fname)

description: This function is used by the integer matrix algebra routines. When entered these routines calls set_fname() with their own function names as arguments. set_fname() will store the current routine's name and if an overflow/underflow is detected by check(), the function name will be put in front of the error message produced by check(). That is, suppose int_mul() is called, then int_mul() makes the call: set_fname(int_mul:). If a overflow occurs inside int_mul(), check() will set the global error message to "int_mul: overflow/underflow". See check() and error() for more details.

arguments: `char *fname`   String containing prefix to error messages produced by `error()`

returns: nothing

usage: set_fname(Function name); /* error messages will look like:
Function name:... */

see also: error(), get_error(), print_error()

file: conv.c

INTTYPE check(INTTYPE result)

description: Checks to if the result of an operation produces an overflow/underflow. When
doing fixed-point arithmetic check() will inspect the current word length to
make sure that result is inside the dynamic range. If an overflow or underflow
occurs, check() will call error() to set the global error string. Pre pended to
the error message is the string set by set_fname(). That is, suppose after the
call set_fname(int_pmul) an overflow is detected by check(). Then the global
error string is set to "in_pmul: overflow/underflow", thereby identifying the
routine which produced the overflow/underflow. The result is returned modulo
the largest number that can be represented by the current word length. That
is, if no overflow or underflow occurs the result is returned unchanged.

arguments: INTTYPE result   Result of a fixed-point operation that is to be checked
for overflow/underflow.

returns: INTTYPE   The returned value is result mod maxint, where maxint is the
largest number representable with the current word length.

usage: INTTYPE a, b, c; c = check(a+b); // c = (a+b) mod maxint, checks for overflow

see also: set_fname(), error(), get_error(), print_error(), init_conv()

file: conv.c

matrix *sub_matrix(matrix *mat, matrix *rows, matrix *cols);

description: Returns the rows and columns of mat that are pointed out by (the matrices) rows and cols. The elements of rows and cols are truncated to integers (with mfloor()) and are used to pick out rows and columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and rows = [3.14 1.99] and cols = [2.1], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(rows, cols). To just index rows, like matlab's mat(rows, :), set cols to NULL. Similarly, to just index columns, like matlab's mat(:, cols), set rows to NULL. By using this scheme all indexing can be done with sub_matrix().

arguments: matrix *mat   Matrix to be indexed.

matrix *rows   Row indexing matrix.

matrix *cols   Column indexing matrix.

returns: matrix *   The indexed input matrix, NULL if error.

usage: B = sub_matrix(mat, rows, cols); // B = mat(rows, cols)

B = sub_matrix(mat, rows, NULL); // B = mat(rows, :)

```
B = sub_matrix(mat, NULL, cols); /* B = mat(:, cols)

matlab equivalent:

>> B = mat(rows, cols); */
```

see also: `index_rows()`, `index_cols()`, `index_rows_cols()`

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: `matrix.c`

matrix *transp(matrix *mat)

description: Returns the transpose of the input matrix. Note: does not conjugate elements. Use herm() for conjugate transpose (hermitian).

arguments: `matrix *mat`   Input matrix.

returns: `matrix *`   The transposed input matrix, NULL if error.

usage: `tran_A = transp(A); /* tran_A = A.'`

```
matlab equivalent:

>> A = A.'; % Note .' not ', that is, does not conjugate */
```

see also: `herm()`

file: `matrix.c`

Pin Number	Signal Name	Description
208—210	BA0-2	Board address.
211—213	PA0-2	Port address.
299	XIW*	X-bus input FIFO write signal.
300	YIW*	Y-bus FIFO write signal.
301	XOR*	X-bus output FIFO read signal.
302	AROE*	Accumulator to X-bus transport enable.
303	XBOE*	X-bus transport output enable.
304	VECTORMODE	Vector mode signal.
305	ROE*	Adder RAM output enable.
306	MROE*	Multiplier RAM output enable.
307	XIFLRT*	X-bus input FIFO restart.
308	XIR*	X-bus input FIFO read signal.
309	YIFLRT*	Y-bus FIFO restart.
310	YIR*	Y-bus FIFO read signal.
311	XOFLRT*	X-bus output FIFO restart.
312	XOW*	X-bus output FIFO write signal.
313	AREN*	Accumulator to X-bus latch enable.
314	ARWE*	Adder RAM write strobe.
315	SREN*	Accumulator latch enable.
316	RESBWE*	Residue bus write enable.
317	RESBRE*	Residue bus read enable.
318	CLR*	Processor wide clear signal.
319	PREN*	Product latch enable.
320	YBEN*	Y-bus transport latch enable.
321	XFEN*	X-bus to multiplier RAM latch enable.
322	XBEN*	X-bus transport latch enable.
323	MRWE*	Multiplier RAM write enable.
324	ARITHMODE	Arithmetic mode signal.

Table 6.1: Gauss Machine Processor Control Signals

Chapter 7

ALGORITHMS

As previously stated, the Gauss machine is designed primarily to perform level 3 operations[11]. It also has a vector mode of operation which handles level 2 and level 1 operations efficiently. Some algorithms for the Gauss machine are described below. Although the Gauss machine operates on GEQRNS operands, the algorithms are discussed using the familiar notation of complex numbers.

7.1 Matrix Product Based Algorithms

This section describes the implementation of the matrix multiplication operation on the Gauss machine and implementation of algorithms which are based upon matrix multiplication.

7.1.1 Matrix Multiplication

If $A \in C^{m \times n}$ and $B \in C^{n \times r}$ then $AB = D$ where $D \in C^{m \times r}$, and each $d_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. The core of the matrix multiplication is the multiply-accumulate operation.

Suppose we wish to multiply two 2×2 matrices. The data are presented in a sloped data front on the X and Y sides of the array and the results are accumulated in place. This configuration is depicted in Figure 7.1. Note that the processors are configured as simple multiply-accumulate units. Each clock cycle, the sloped A and B data fronts are advanced one processing element to the right and up, respectively,

and the input operands are multiplied and accumulated.

The configuration shown in Figure 7.1 assumes that the array elements are pre-initialized to zero. The leading zeros keep the array initialized to zero and the trailing zeros maintain the results while the computation is completed. This multiplication uses 2×2 input matrices, however, it can be extended to input matrices where $\mathbf{A}$ is $2 \times n$ and $\mathbf{B}$ is $n \times 2$. Matrix multiplication of larger arrays may be achieved by using block multiplication. Systolic arrays which use in-place accumulation of results are well suited to block operations.
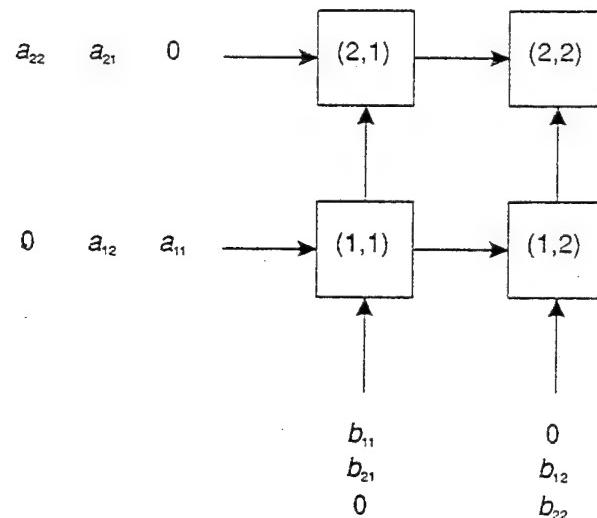
Figure 7.1: Example of Matrix Multiplication

The computation of $\mathbf{AB}$ where $\mathbf{A}$ and $\mathbf{B}$ are of arbitrary dimensions can be performed by decomposing $\mathbf{A}$ into blocks of two rows and n columns, and by decomposing $\mathbf{B}$ into blocks of n rows and two columns. This decomposition is depicted

below:

$$A = \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ \vdots \\ A_i \end{pmatrix}$$

$$B = \begin{pmatrix} B_1 & B_2 & B_3 & \cdots & B_j \end{pmatrix}.$$

where each $A_k \in C^{2 \times n}$ and each $B_l \in C^{n \times 2}$. If a matrix does not have dimensions which meet the multiple of three rows or columns requirements, then the matrix may be padded with zeros. Each of these matrix block products requires $n + 4$ cycles to complete the computation and two cycles to remove the results from the array. The result of the matrix block computation is a 2×2 matrix. Thus for $A \in C^{k \times n}$ and $B \in C^{n \times r}$, the number of cycles required to complete the product AB is given by

$$O(AB) = \lceil k/2 \rceil \lceil r/2 \rceil (n + 6), \tag{7.1}$$

where $\lceil \bullet \rceil$ represents the greatest integer or ceiling function. Clearly from Equation 7.1, if n is small then the overhead associated with the time required to shift data out and the zero padding becomes significant. As $n \to 1$, the operation degenerates into an outer product. The dynamic range requirements for the matrix multiplication are determined by the size of A and B, and the dynamic range of the data in the matrices. To be exact, suppose that A has a dynamic range of p, B has a dynamic range of q, and the dimension quantity n is as given above. Then the dynamic range requirements for the product AB is given by

$$O_D(AB) = (p + q) + n - 1. \tag{7.2}$$

7.1.2 Discrete Fourier Transform

The DFT may be easily expressed as a level 3 operation. The DFT is given by the following:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}.$$

We may express this function in a linear algebraic form by assigning all $x(n)$ to a column vector, $\mathbf{x} \in \mathbf{C}^{N \times 1}$, and the complex exponential to a Vandermonde matrix $\mathbf{W} \in \mathbf{C}^{N \times N}$ where $\mathbf{W}$ is given by

$$\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ W^{-1} & W^{-2} & W^{-3} & \cdots & W^{-(N-1)} \\ W^{-2} & W^{-4} & W^{-6} & \cdots & W^{-2(N-1)} \\ W^{-3} & W^{-6} & W^{-9} & \cdots & W^{-3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ W^{-(N-1)} & W^{-2(N-1)} & W^{-3(N-1)} & \cdots & W^{-(N-1)^2} \end{pmatrix},$$

where $W = e^{j2\pi/N}$. Thus the DFT of $\mathbf{x}$ is given by

$$\mathbf{X} = \mathbf{W}^T \mathbf{x}$$

Where $\mathbf{X} \in \mathbf{C}^{N \times 1}$. Clearly $\mathbf{W}$ may be precomputed and its form depends upon the number of points to be computed and the dynamic range requirements of the problem. Obviously, since the Gauss machine performs level 3 operations at greatest efficiency when the dimensions of the operand matrices are multiples of two it may be desirable to gang three DFTs together by replacing the column vector $\mathbf{x}$ with an $N \times 2$ matrix where each column of data represents a vector to be transformed. Additionally, if only selected frequency bands are of interest then the DFT need only be computed for those bands.

7.1.3 Convolution and Correlation

Like the DFT, convolution and correlation may be expressed as level 3 operations. Recall that discrete correlation and convolution are given by

$$X(\tau) = \sum_n x(n)y(n + \tau),$$
$$Y(\tau) = \sum_n x(n)y(\tau - n),$$

respectively. We will develop the correlation example here; convolution follows directly. Let x and y be data sequences of length N and M, respectively. The correlation of x and y will have a length of $N + M - 1$. We will assign the sequence x to a row vector $\mathbf{x} \in \mathbf{C}^{1 \times (N+M-1)}$ given by

$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 & x_2 & \cdots & x_{N-1} & | & 0 & \cdots & 0 \end{pmatrix}.$$

The sequence y is used to build a matrix $\mathbf{Y} \in \mathbf{C}^{N+M-1 \times N+M-1}$, which when multiplied with $\mathbf{x}$ will give the correlation sequence of x and y. Since $\mathbf{x}$ is a row vector we would like the columns of $\mathbf{Y}$ to be shifted versions of the sequence y. Assuming $y = \{y_0, y_1, y_2, \ldots, y_{M-1}\}$, then the matrix $\mathbf{Y}$ would have the form

$$\mathbf{Y} = \begin{pmatrix} y_{M-1} & y_{M-2} & y_{M-3} & \cdots & 0 & 0 & 0 \\ 0 & y_{M-1} & y_{M-2} & \cdots & 0 & 0 & 0 \\ 0 & 0 & y_{M-1} & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & y_1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & y_2 & y_1 & 0 \\ 0 & 0 & 0 & \cdots & y_3 & y_2 & y_1 \end{pmatrix}.$$

Thus, we can see that the product $\mathbf{xY}$ will produce the correlation sequence of x and y. The above form of the correlation is suboptimal in that it uses level 2 operations to perform the correlation operation. Optimal performance is obtained by utilizing level

3 operations to perform the correlation. We will form a new matrix $X \in \mathbf{C}^{2 \times N+M-1}$ which contains a shifted version of the row vector x:

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & \cdots & x_{N-1} & 0 & 0 & \cdots & 0 \\ 0 & x_0 & x_1 & x_2 & \cdots & & x_{N-1} & 0 & \cdots & 0 \end{pmatrix}.$$

The form of the matrix X requires that the form of Y be modified so that it reflects the shifts in X. The first column of the new version of Y, Y', will produce the first three elements of the correlation sequence; the second column will produce the next three, and so forth. The form of Y' is given as

$$Y' = \begin{pmatrix} y_{M-3} & y_{M-6} & \cdots & 0 & 0 \\ y_{M-2} & y_{M-5} & \cdots & 0 & 0 \\ y_{M-1} & y_{M-4} & \cdots & 0 & 0 \\ 0 & y_{M-3} & \cdots & 0 & 0 \\ 0 & y_{M-2} & \cdots & 0 & 0 \\ 0 & y_{M-1} & \cdots & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \cdots & y_1 & 0 \\ 0 & 0 & \cdots & y_2 & 0 \\ 0 & 0 & \cdots & y_3 & 0 \\ 0 & 0 & \cdots & y_4 & y_1 \\ 0 & 0 & \cdots & y_5 & y_2 \\ 0 & 0 & \cdots & y_6 & y_3 \end{pmatrix}.$$

The product has the form $XY' \in \mathbf{C}^{2 \times \lceil (N+M-1)/2 \rceil}$. This level 3 form of the correlation operation executes two times faster than the level 2 version, and the computational speed of the level 3 algorithm will grow geometrically with N for an $N \times N$ array, while the level 2 version will only have a linear growth of the computational speed on the same array. Additionally, since Y' is banded the structure can be exploited to avoid mass multiplication by zeros.

7.2 Vector Mode Algorithms

While many algorithms may be expressed as level 3 operations, there are several operations which may not be performed efficiently on the Gauss machine while operating in systolic mode. In order to improve this situation, an architectural enhancement was made to allow the Gauss machine to operate in vector mode. The vector mode of operation uses a subset of the Gauss machine's processing elements and a minimal amount of additional hardware to form the vector processor. See Figure 6.3 and Figure 6.4.

7.2.1 Vector Addition

A common operation is the addition of two vectors. While the systolic mode of operation can be used accumulate two vectors together, it is very inefficient. Let $x, y \in C^N$ be two vectors which are to be added. In order to add them in the systolic mode of operation of the array, they are appended so as to form a matrix $Z \in C^{N \times 2}$ such that

$$Z = (x|y).$$

The matrix Z is then multiplied by the matrix (11) to form the sum of the two vectors:

$$x + y = (11)Z.$$

This technique may be extended to the accumulation of K vectors, leading to the form

$$\sum_{i=0}^{K-1} x_i = \underbrace{(111 \cdots 1)}_{K} (x_0|x_1|x_2| \cdots x_{K-1}).$$

From Equation 7.1 we know that the number of cycles required to complete the product is

$$\lceil K/2 \rceil (N + 6),$$

which produces a total of $(K - 1) N$ complex addition operations. This leads to an efficiency metric for systolic mode vector addition given by

$$\eta_{AS} = \frac{(K - 1) N}{\lceil K/2 \rceil (N + 6)} \text{complex additions per cycle.}$$

In the vector mode of addition, the summands are accumulated after being multiplied by one. The result is that the number of cycles required for the addition of K vectors of length N is given by

$$KN + 3,$$

thus leading to an efficiency metric of

$$\eta_{AV} = \frac{2 (K - 1) N}{KN + 3} \text{complex additions per cycle,}$$

where the factor of 2 is a result of operating two PEs in tandem as vector accumulators. For large N, and K even, $\eta_{AS} \approx \eta_{AV}$. However, for N large and K odd, then

$$\frac{\eta_{AV}}{\eta_{AS}} = \frac{K + 1}{K}.$$

In general, we would expect the vector method to have slightly better performance than the systolic method.

7.2.2   Pointwise Vector Multiplication

Pointwise multiplication of vectors is an important operation in signal processing applications. Pointwise multiplication of vectors is used to window vectors, and to

scale a vector. In systolic mode, a vector of length N may be pointwise multiplied by another vector by taking the second vector and placing it on the diagonal of an $N \times N$ matrix of zeros and then multiplying it with the first vector as follows. Suppose $\mathbf{x}, \mathbf{y} \in \mathbf{C}^N$. Then the pointwise vector product (denoted by $.*$) of these two vectors is

$$\mathbf{x}.*\mathbf{y} = \begin{pmatrix} x_0 & 0 & 0 & \cdots & 0 \\ 0 & x_1 & 0 & \cdots & 0 \\ 0 & 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & x_{N-1} \end{pmatrix} \mathbf{y}.$$

This expression may be evaluated in a manner which reduces the obvious large order of computation by just producing the block matrix products of the vector and the 2×2 submatrices centered along the diagonal. This produces an expression for the number of cycles required to execute the pointwise vector product given as $\lceil N/2 \rceil 8$ cycles. Thus, since this is the number of cycles required to execute N multiplications, the efficiency metric is given as

$$\eta_{MS} = \frac{N}{\lceil N/2 \rceil 8} \text{complex multiplications per cycle.}$$

In vector mode, the two vector PEs may multiply pairs of multiplicands in a pipelined manner, one product per cycle per PE. Thus, it requires $\lceil N/2 \rceil$ cycles to pointwise multiply two vectors of length N. This leads to an efficiency metric of

$$\eta_{MV} = \frac{N}{\lceil N/2 \rceil} \text{complex multiplications per cycle.}$$

For large N, the vector mode of operation is approximately eight times faster than the systolic mode of operation.

7.3 QR Decomposition

An important application of the RNS is the solution of linear equations of the form

$$Ax = b.$$

Methods such as Gaussian elimination, while amenable to implementation on a vector machine, are not as robust as the QR decomposition (QRD). The QR decomposition is given below.

Theorem 2 (QR Factorization) *If* $A \in C^{m \times n}$ *of rank n, then* A *can be factored into a product* QR*, where* $Q \in C^{m \times n}$*, and is a matrix with orthonormal columns, and* $R \in C^{n \times n}$ *and is upper triangular and invertible.*

The (QRD) can be performed efficiently in the RNS using the Householder reflection. The Householder reflection is preferable over the Givens rotation since it does not contain transcendental functions. Since the RNS is a division-free integer system of arithmetic, it cannot compute transcendental functions efficiently. Additionally, Givens rotations suffer from potential finite-precision error conditions to which the Householder reflections can be made immune. Finally, the Householder reflection is inner-product rich, thus making it ideal for Gauss machine implementation.

7.3.1 Householder Reflections

The Householder reflection is an orthogonal vector transformation which is used to selectively introduce zeros in a vector by reflecting the vector through a plane. In general, the Householder reflection is very efficient when used to introduce a large number of zeros into a vector.

The Householder reflection introduces zeros into a vector $x \neq 0$ via an orthogonal transformation. The transformation matrix, $\mathbf{H}$ is defined by

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}},$$

where $\mathbf{v}$, the Householder vector is defined by

$$\mathbf{v} = \mathbf{x} \pm \|\mathbf{x}\|_2 \, e_1. \tag{7.3}$$

When the Householder matrix is applied to the vector $\mathbf{x}$, we arrive at

$$\mathbf{H}\mathbf{x} = \left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}\right)\mathbf{x} = \mp \|\mathbf{x}\|_2 \, e_1.$$

In computing the Householder vector, $\mathbf{v}$, we may choose to use either of the forms

$$\mathbf{v} = \mathbf{x} + \|\mathbf{x}\|_2 \, e_1, \text{ or}$$
$$\mathbf{v} = \mathbf{x} - \|\mathbf{x}\|_2 \, e_1.$$

It is desirable to keep the 2-norm of the Householder vector from becoming small since it would result in the scalar vector $2/\mathbf{v}^T\mathbf{v}$ from Equation 7.3 having a large relative error. Thus we may choose $\mathbf{v}$ in such a way as to maximize the 2-norm of $\mathbf{v}$:

$$\mathbf{v} = \mathbf{x} + \text{sign}\,(x_1)\,\|\mathbf{x}\|_2 \, e_1.$$

Since the RNS is a division-free system of arithmetic, reduction of division operations is attractive. Usually, when the Householder transform is used, it is typically used *en masse*. Thus, it is desirable to maintain the individual Householder transform matrix as a quotient of the form

$$\frac{\mathbf{v}^T\mathbf{v}\mathbf{I} - 2\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}},$$

leading to the form

$$\mathbf{H} \longleftrightarrow \left(\mathbf{v}^T \mathbf{v}, \mathbf{H}' \right),$$

where

$$\begin{aligned} \mathbf{H}' &= \mathbf{v}^T \mathbf{v} \mathbf{H} \\ &= \mathbf{v}^T \mathbf{v} \mathbf{I} - 2\mathbf{v}\mathbf{v}^T. \end{aligned} \tag{7.4}$$

This form avoids division, at the expense of consumption of dynamic range. In particular, if $\mathbf{x}$ is of length N, and the dynamic range of the elements of $\mathbf{x}$ is p, then the dynamic range of $\mathbf{v}$ is $(N+1)p$. From this we have the dynamic range of $\mathbf{v}^T\mathbf{v}$ given by $(N+1)p + N - 1$. Assuming that $N > 2$, then we may approximate the dynamic range of $\mathbf{H}'$ as $(N+1)p + N - 1$. The next section discusses the application of the Householder transform to the problem of the QRD.

7.3.2 Householder QR Factorization

The previous section introduced the use of the Householder transform for the introduction of zeros into a vector. By repeated application of the Householder transformation, we may decompose a matrix $\mathbf{A}$ into an orthogonal matrix and an upper-right triangular matrix, as discussed in Theorem 2 in Section 7.3. The following discussion will examine the implementation of the QRD using the Householder transformation, with special emphasis on the implications of using the transform on the Gauss machine.

Suppose we have $\mathbf{A} \in \mathbf{R}^{m \times n}$, and wish to produce the QR decomposition of $\mathbf{A}$. Then we must generate a series of orthogonal transforms which, when applied to $\mathbf{A}$, will reduce $\mathbf{A}$ to upper-right triangular form. Define a series of orthogonal

transforms

$$\tilde{H}_i = \begin{bmatrix} I_i & 0 \\ 0 & H'_i \end{bmatrix},$$

where I_i is the identity matrix of order i, and H'_i is of the form of Equation 7.4,

$$H'_i = v^{(i)T}v^{(i)}I - 2v^{(i)}v^{(i)T}, \tag{7.5}$$

where $v^{(i)}$ is defined to be the sub-diagonal entries of the ith column of A. Thus, we may successively apply the $\tilde{H}_i$'s to A:

$$\tilde{H}_n \tilde{H}_{n-1} \tilde{H}_{n-2} \cdots \tilde{H}_1 A,$$

and thus,

$$A = QR,$$

where

$$Q = \frac{\tilde{H}_n \tilde{H}_{n-1} \tilde{H}_{n-2} \cdots \tilde{H}_1}{\prod_{i=1}^{n} v^{(i)T}v^{(i)}}.$$

7.3.3 Dynamic Range Requirements of the Householder QRD

Since an overflow condition cannot be detected in an RNS system, it is necessary to determine the dynamic range requirements of a given algorithm before it may be used with confidence. Computations must proceed assuming a worst-case set of input data, unless, some occasional error is acceptable.

As in the previous section, suppose we have a matrix $A \in R^{m \times n}$, and we truncate A to some finite precision where we may represent each a_{ij} with k or fewer bits, for all $i \in \{1, 2, 3, \ldots, m\}$, $j \in \{1, 2, 3, \ldots, n\}$.

To determine H'_i we examine Equation 7.5. The first term contains an inner product which acts as a coefficient to an identity matrix. The vector $v^{(i)}$ is determined

using Equation 7.3. The dynamic range of $v_1^{(i)}$ is $k+1$ bits while $v_j^{(i)}$ has a dynamic range of k bits for all $j \in \{2, 3, 4, \ldots, (m-i)\}$. Thus the dynamic range of the inner product of Equation 7.5 is $2(k+1) + \lceil (m-i-1)/3 \rceil$.

Proof:

Let $\mathbf{v} \in \mathbf{R}^n$. Then

$$\mathbf{v}^T\mathbf{v} = v_1^2 + v_2^2 + v_3^2 + \cdots + v_n^2,$$

where v_1 is known to have a dynamic range of $k+1$ bits, and all v_i have a dynamic range of k bits, for $i \in \{1, 2, 3, \ldots, n\}$. Thus, $O\left(v_1^2\right) = 2(k+1)$ and $O\left(v_i^2\right) = 2k$. Examining the above summation, we see that the dynamic range is easily computed:

$$\overbrace{v_1^2}^{2k+2} + \underbrace{\overbrace{v_2^2}^{2k} + \overbrace{v_3^2}^{2k}}_{\underbrace{2k+1}} + \overbrace{v_4^2}^{2k} + \cdots .$$

This leads to a dynamic range bound of $2(k+1) + \lceil (n-1)/3 \rceil$.

The second term of Equation 7.5 is a scaled outer product. Let $\mathbf{B} = 2\mathbf{v}^{(i)}\mathbf{v}^{(i)T}$. Then b_{11} has dynamic range $2(k+1) + 1 = 2k+3$ while b_{1p} and b_{j1} have dynamic range $(k+1) + k + 1 = 2(k+1)$, and b_{jp} has dynamic range $k + k + 1 = 2k+1$, for all $j \neq 1$, and all $p \neq 1$. To summarize these findings, the dynamic range of $\mathbf{B}$ is given as

$$O_D(\mathbf{B}) = \begin{pmatrix} 2k+3 & 2(k+1) & \cdots & 2(k+1) \\ 2(k+1) & 2k+1 & \cdots & 2k+1 \\ \vdots & \vdots & \ddots & \vdots \\ 2(k+1) & 2k+1 & \cdots & 2k+1 \end{pmatrix}. \tag{7.6}$$

Finally, the two terms of Equation 7.5 are subtracted leading to the final result for the dynamic range of $\mathbf{H}_i'$. Clearly, since the first term is a scaled identity matrix, the dynamic range of the off-diagonal elements will be as given in Equation 7.6. For

$h'_{i_{11}}$, if $(m - i) \leq 3$ then the dynamic range is $2k + 4$, else, if $(m - i) > 3$ then the dynamic range is $2(k + 1) + \lceil (m - i)/3 \rceil$. For the remaining diagonal elements, $h'_{i_{jj}}$, $j \neq 1$, the dynamic range is $2(k + 1) + \lceil (m - i)/3 \rceil$. To summarize,

$$
O_D\left(\mathbf{H}'_i\right) = \left(
\begin{array}{c|ccccc}
a_i & 2(k+1) & \cdots & & \cdots & 2(k+1) \\
\hline
2(k+1) & b_i & 2k+1 & \cdots & & 2k+1 \\
\vdots & 2k+1 & \ddots & & \ddots & \vdots \\
\vdots & \vdots & & \ddots & \ddots & 2k+1 \\
2(k+1) & 2k+1 & \cdots & & 2k+1 & b_i
\end{array}
\right),
$$

where

$$
a_i = \begin{cases} 2k + 4 & (m - i) \leq 3 \\ 2(k + 1) + \lceil (m - i)/3 \rceil & (m - i) > 3 \end{cases},
$$

and $b_i = 2(k + 1) + \lceil (m - i)/3 \rceil$.

Clearly, the dynamic range of the above approach quickly gets out of hand. An alternative approach is suggested by [11]. This approach relies on a block representation and is given below.

```
Y = v^(1)
W = -2v^(1)/v^(1)T v^(1)
for j=2:r
    z = -2(I + WY^T)v^(j)/v^(j)T v^(j)
    W = [Wz]
    Y = [Yv^(j)]
end
```

Let the Householder vectors $\mathbf{v}^{(i)}$ be used to pre-generate $\hat{\mathbf{v}}^{(i)} = -2\mathbf{v}^{(i)}/\mathbf{v}^{(i)T}\mathbf{v}^{(i)}$. Then the above algorithm is modified to take the form given below.

```
Y = v^(1)
W = v̂^(1)
for j=2:r
    z = (I + WY^T)v̂^(j)
    W = [Wz]
    Y = [Yv^(j)]
end
```

Clearly the modified algorithm is relatively rich in level 3 operations. As the index j increases, the order of computation for the outer product $\mathbf{W}\mathbf{Y}^T$ increases as the number of columns of $\mathbf{W}$ and $\mathbf{Y}$ increases, and thus the processor utilization increases.

Chapter 8

SUMMARY AND CONCLUSIONS

8.1   Motivation

There is a demonstrable need for high speed front-end signal processors for signal
and image processing applications. There exist a number of problems (*e.g.*, RADAR,
communications, video processing) which demand a level of performance which ex-
ceeds the capabilities of the current generation of DSP microprocessors by an order
of magnitude or more. This need for speed cannot exacerbate existing constraints
on size, power, reliability, and cost. With this motivation the construction of a pro-
totype array of processors based upon the GEQRNS was undertaken. The goal of
this prototype was to demonstrate that an array of RNS-based processors could be
used to obtain high computational throughputs without exacerbating the aforemen-
tioned problems. The Gauss machine was constructed using discrete components as
a prototype to a VLSI implementation.

Another goal of the Gauss machine was to demonstrate that an array processor
could be constructed which would be applicable to a rich set of problems and thus
demonstrate that a technology which was sub-optimal in the architectural sense could
be used for a variety of problems, thus affecting a savings in non-recurring engineering
costs (NREs). Most signal processing problems are rich in inner products which may
be expressed in terms of level 1, level 2, or level 3 operations. The Gauss machine
was designed to achieve a high level of efficiency in performing level 3 operations, and

a lesser, although still excellent, throughput in level 2 and level 1 operations. Many signal processing problems can be efficiently stated in terms of level 3 operations, and thus this bias towards level 3 operations was created.

8.2   Results

There were a number of problems which had to be solved in order to construct the Gauss machine. Initially, certain problems in developing an experimental environment were identified. These problems included packaging constraints, flexibility, and portability. Packaging constraints were solved by developing our own prototyping environment, the InvestiGATOR. The InvestiGATOR also solved the problems of portability and flexibility by incorporating a general purpose computer based upon the Motorola 68030 microprocessor, and the inclusion of the SCSI interface for high speed, portable communications, and the RS-232C interface for low speed, portable communications. The InvestiGATOR solved mechanical packaging constraints encountered in earlier efforts by the construction of a large backplane based system. The backplane includes broadcast and near-neighbor communications making it suitable for a variety of prototyping tasks. Additionally, mechanical constraints were reduced to two dimensions (thickness and length) from the three dimensions (thickness, length, and height) found in conventional environments.

The Gauss machine was constructed on six boards which reside on the InvestiGATOR backplane. Each of these six boards has a 2×2 array of seven-bit GEQRNS processors. These processors were constructed using low-cost, commodity discrete logic components. The arithmetic elements were implemented with low-cost $32K \times 8$ 85 ns SRAM. The arithmetic elements (SRAMs) are the limiting factor in the speed of the system: the 85 ns SRAMs are suitable for a 10 MHz clock rate, while 35 ns

SRAMs would be suitable for a greater than 20 MHz clock rate, and 15 ns SRAMs could be used to generate a clock rate of 50 MHz. At the 10 MHz clock rate, the array achieves an equivalent peak arithmetic rate of 320 million operations per second when performing complex arithmetic, compared with conventional processors. Each of the processor elements on the board occupies approximately $4.4\,in^2$ of board area and is a discrete implementation of a structure which occupies only approximately $2\,mm^2$ when implemented in the MOSIS 2.0 μm scalable CMOS process.

The six processor boards may be configured to act as a single GEQRNS array which can process arithmetic word widths of approximately twenty-one bits (20.25 bits, or 122 dB). Alternately, the Gauss machine may be configured as a single conventional RNS array processor with a dynamic range of approximately thirty-three bits. Additionally, the array processor may be configured to operate as a vector processor using a subset of the processing elements in the array. While the whole array is ideal for level 3 operations, it cannot perform level 2 and level 1 operations efficiently. To solve this problem, a vector sub-processor was carved out of the array. The vector sub-processor can be used to efficiently perform level 2 and level 1 operations such as matrix-vector, and vector-vector inner products, as well as pointwise addition and multiplication. In the vector processor mode, the peak arithmetic rate is equivalent to 160 million operations per second when performing complex arithmetic, compared to conventional processors.

One of the original goals of the project that was dropped due to budgetary constraints was the construction of forward conversion and CRT engine in hardware. The hardware implementation was replaced by a high performance software implementation which runs on the InvestiGATOR communication processor. While this loss does prevent the Gauss machine from being used for high speed real-time appli-

cations, it does not seriously interfere with the goals to be demonstrated since the Gauss machine processor array does demonstrate high arithmetic rates.

In conclusion, the Gauss machine demonstrates a high performance, high RNS content architecture for signal processing applications. The Gauss machine performs at an equivalent peak processing rate of 320 million operations per second when performing complex arithmetic, compared to conventional processors. The Gauss machine demonstrates fault tolerance at an architectural level due to the properties of the RNS. This discrete implementation of the Gauss machine demonstrates a cost parity with conventional, off the shelf technologies, however, substantial cost savings can be expected in a VLSI version of this technology, even when produced for relatively short production runs. The Gauss machine also demonstrates an architecture which can potentially be scaled into other technologies (*e.g.*, ECL and GaAs) to produce performance which exceeds that of the Gauss machine by an order of magnitude or more, thus yielding performance several orders of magnitude greater than that possible with conventional signal processing technology.

Appendix A

INVESTIGATOR SCHEMATICS
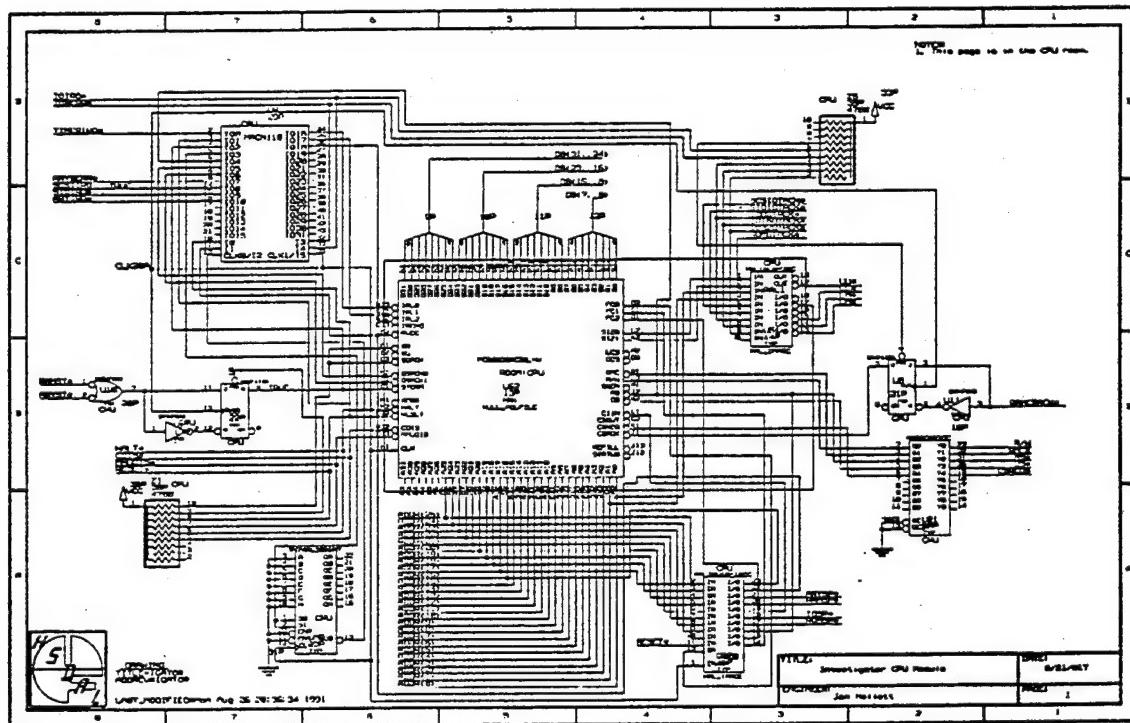
Figure A.1: InvestiGATOR CPU Module

140
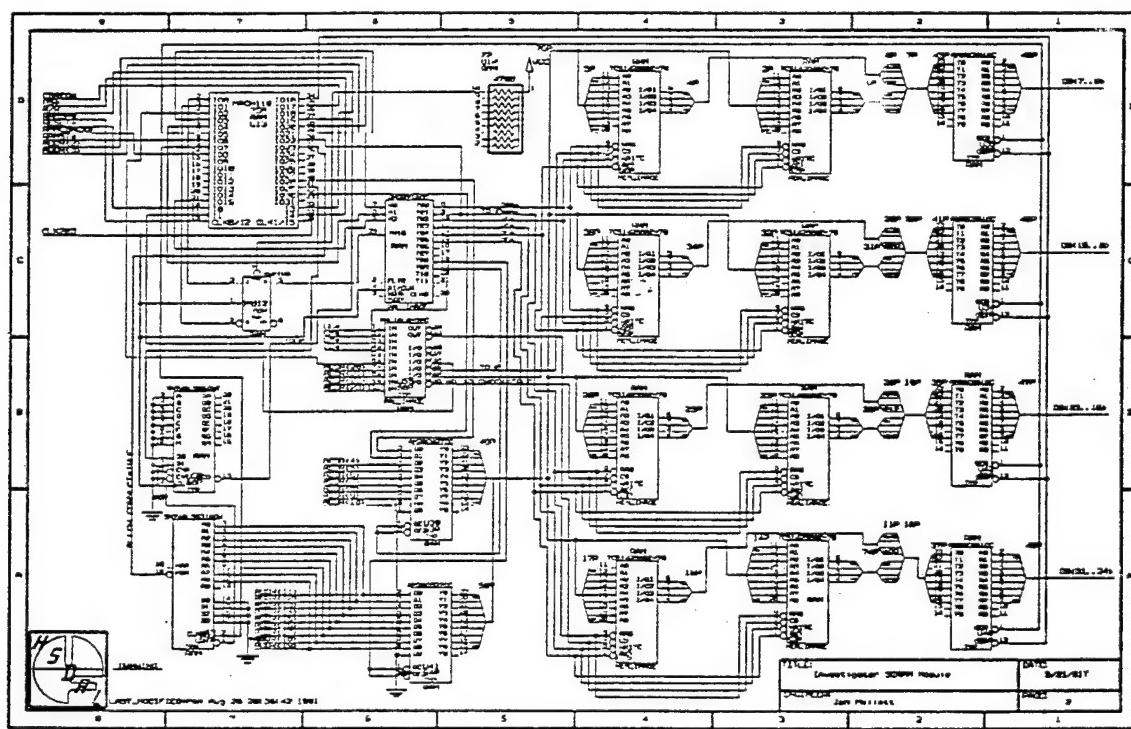
Figure A.2: InvestiGATOR SCRAM Module
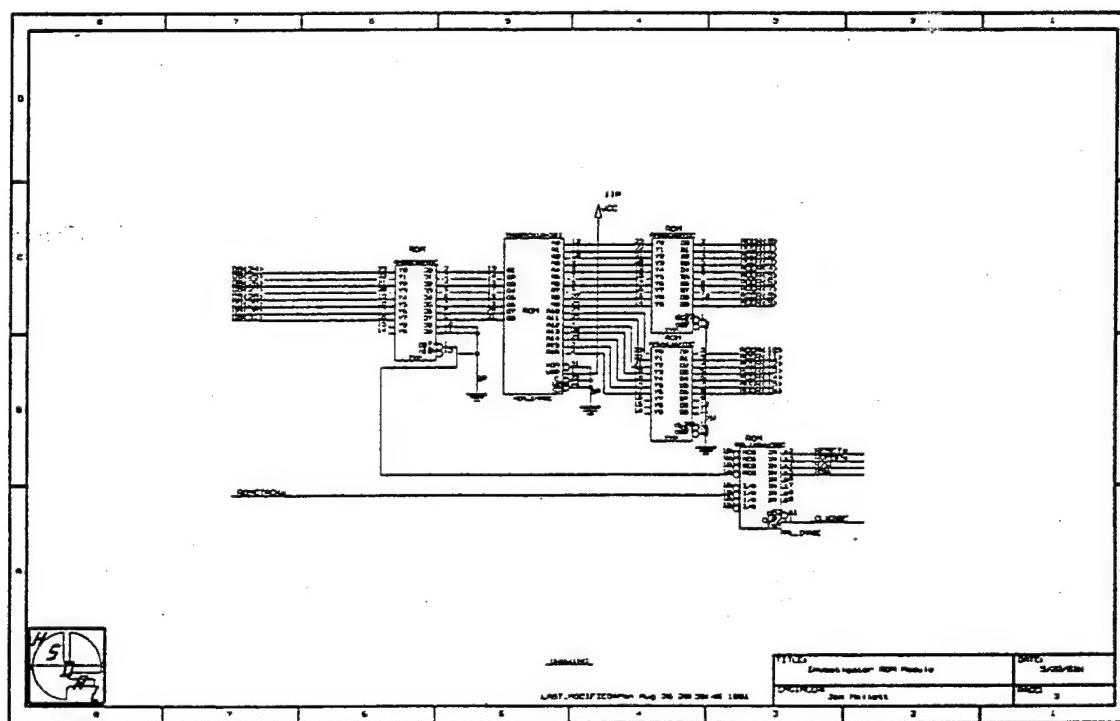
Figure A.3: InvestiGATOR ROM Module

142
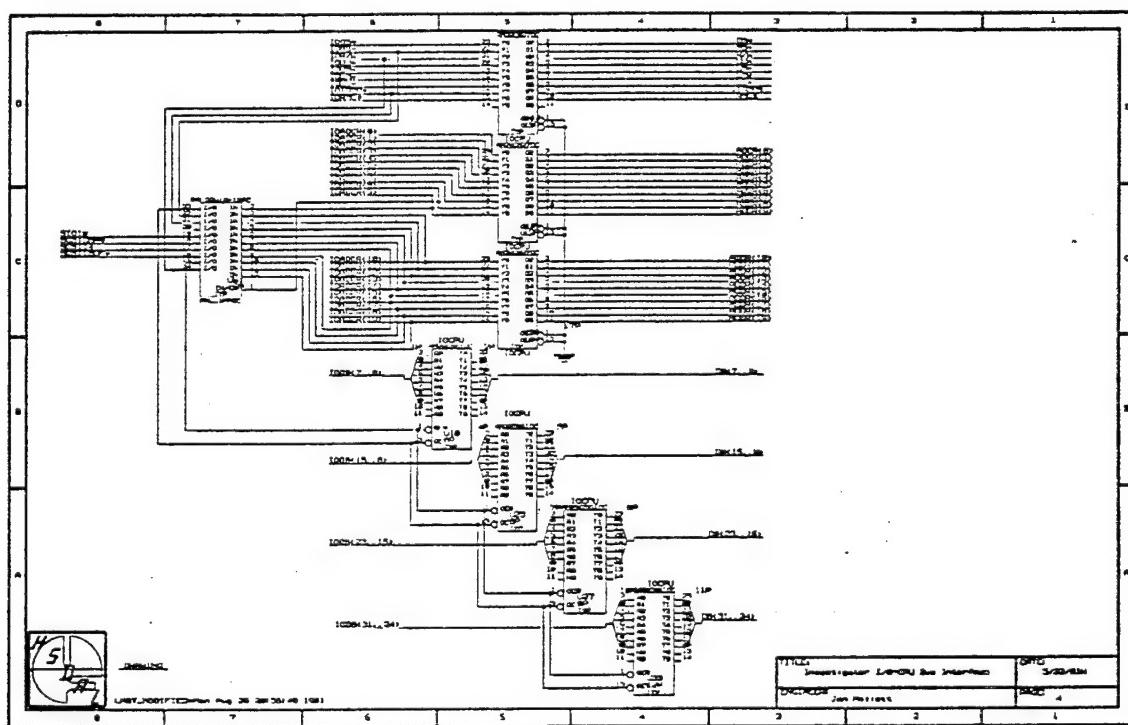
Figure A.4: InvestiGATOR I/O Module

Figure A.5: InvestiGATOR SCSI Module
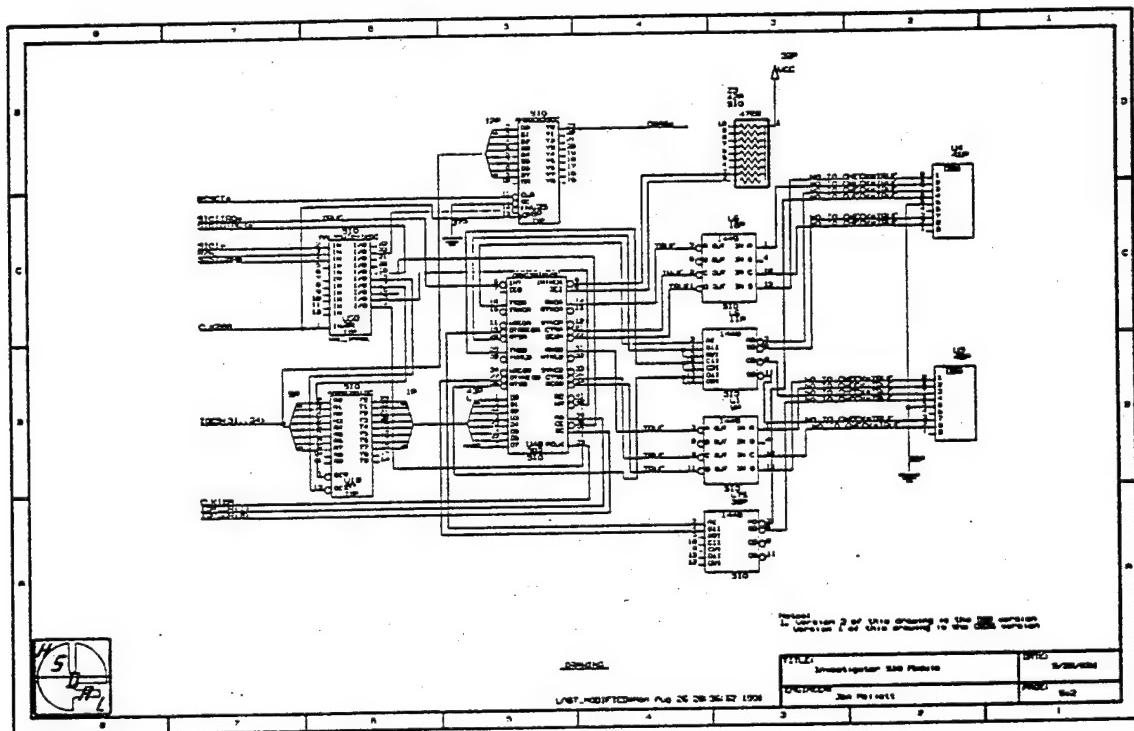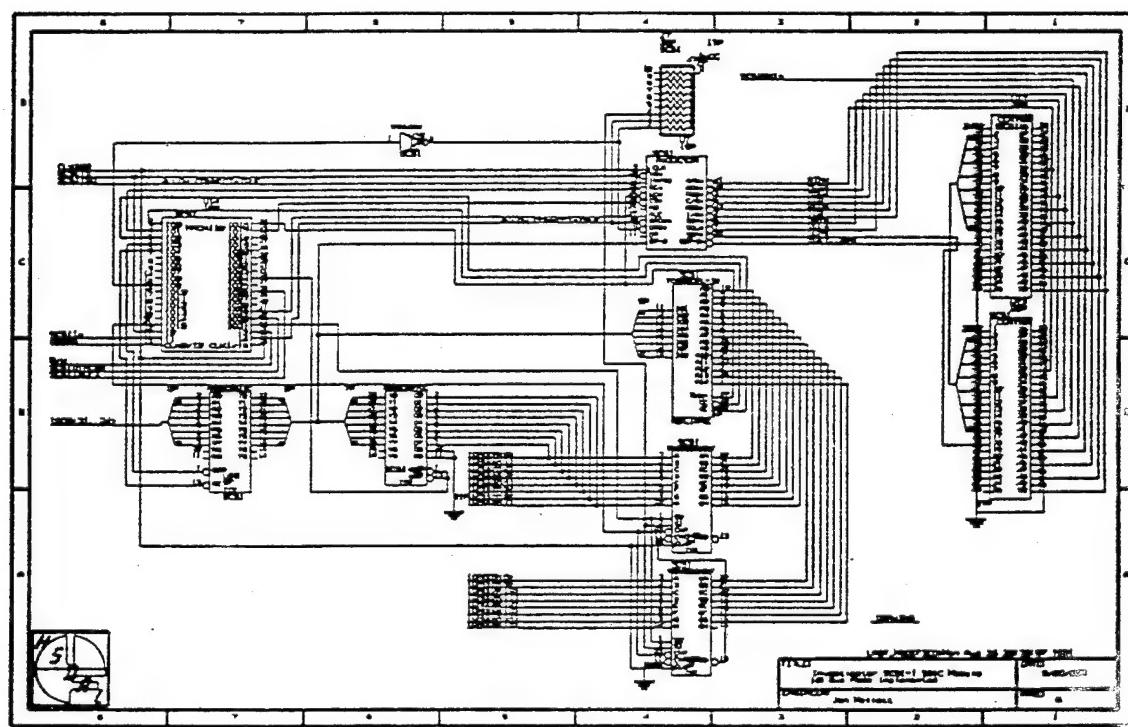
Figure A.6: InvestiGATOR SIO Module
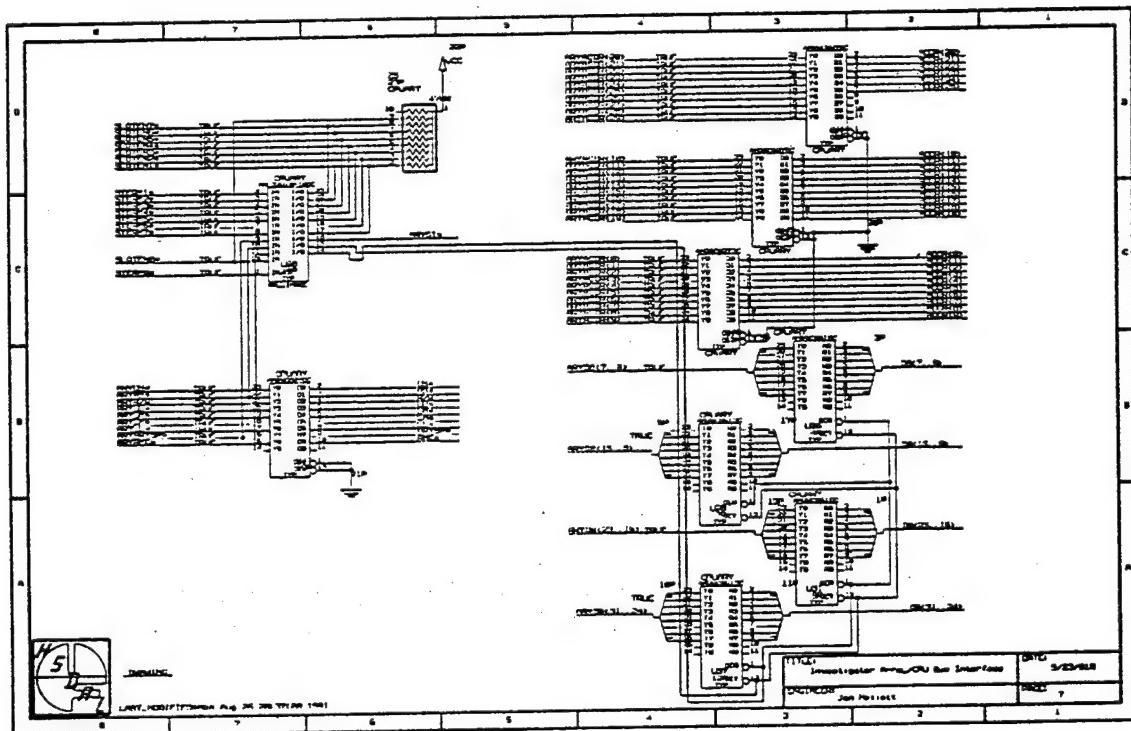
Figure A.7: InvestiGATOR Array Bus Interface
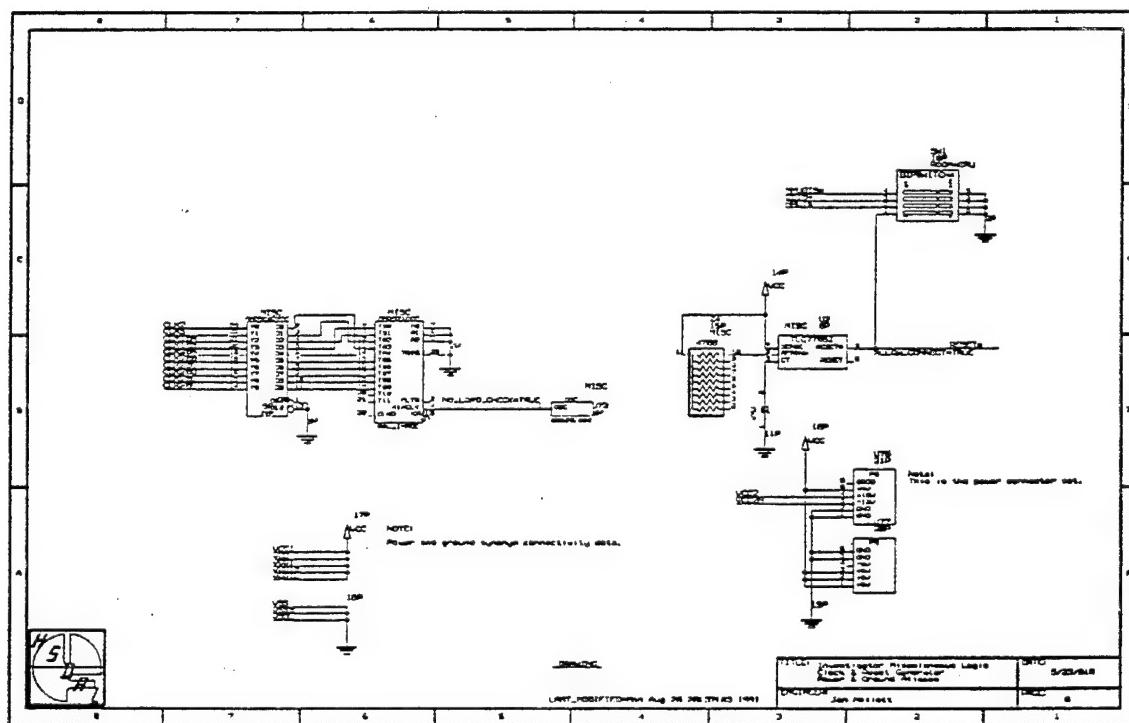
Figure A.8: InvestiGATOR MIscellaneous Module
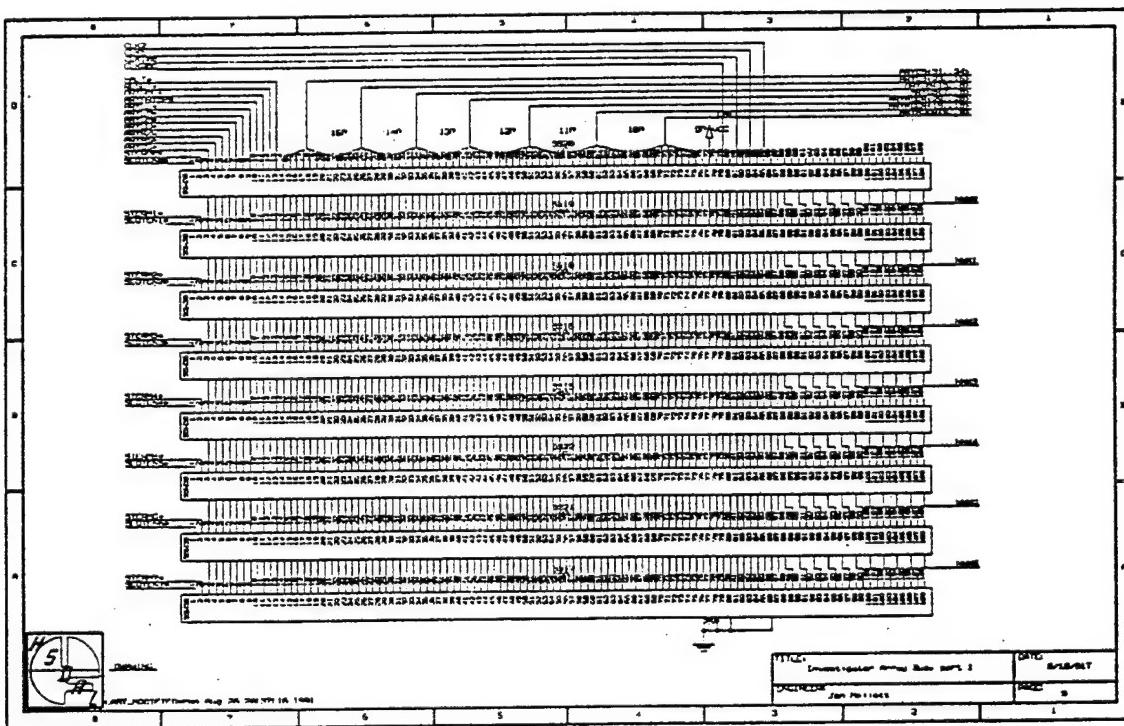
Figure A.9: InvestiGATOR Array Bus, First Part
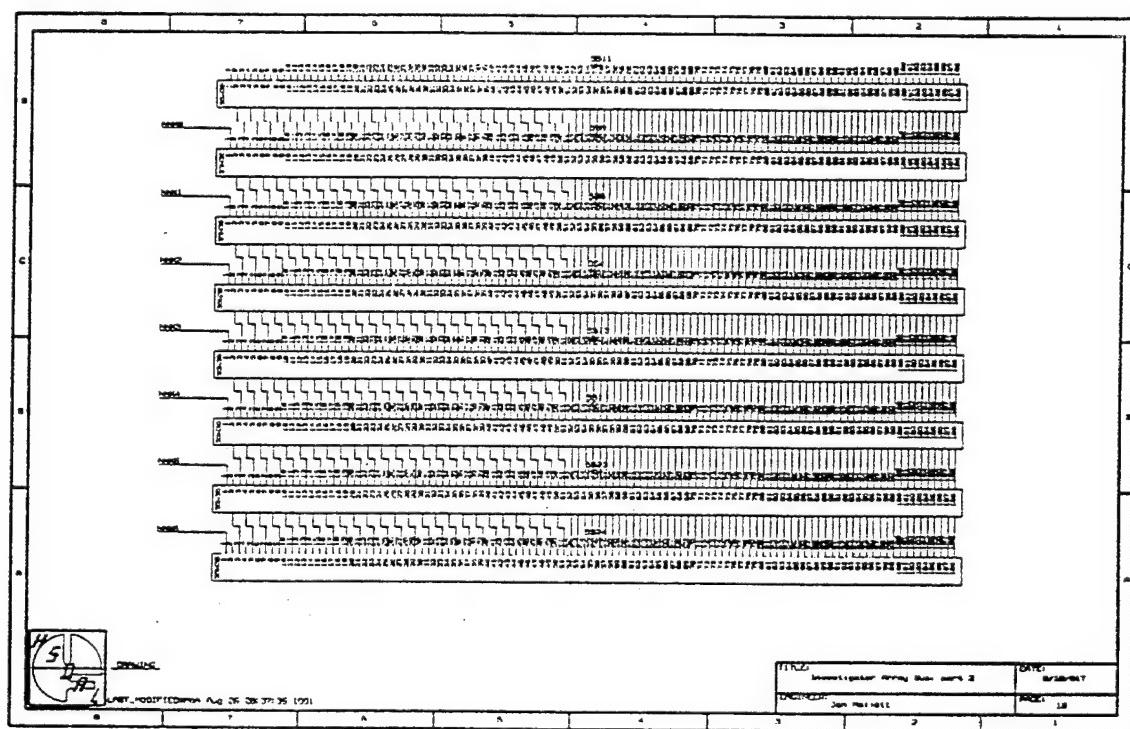
Figure A.10: InvestiGATOR Array Bus, Second Part
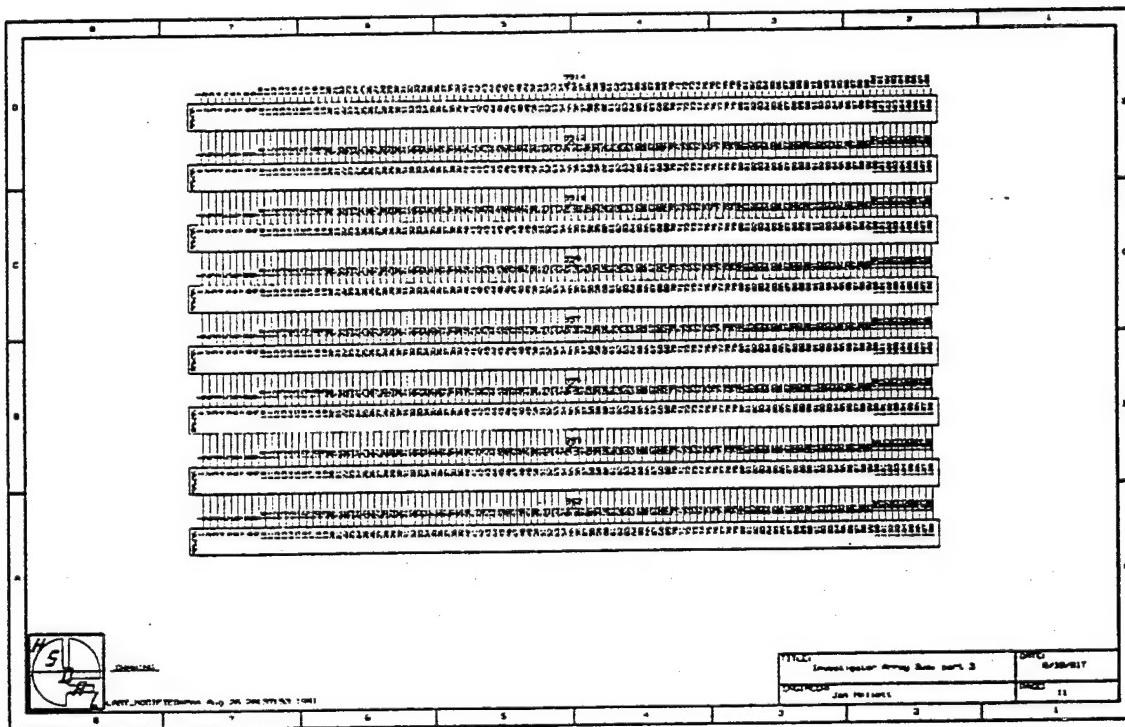
Figure A.11: InvestiGATOR Array Bus, Third Part
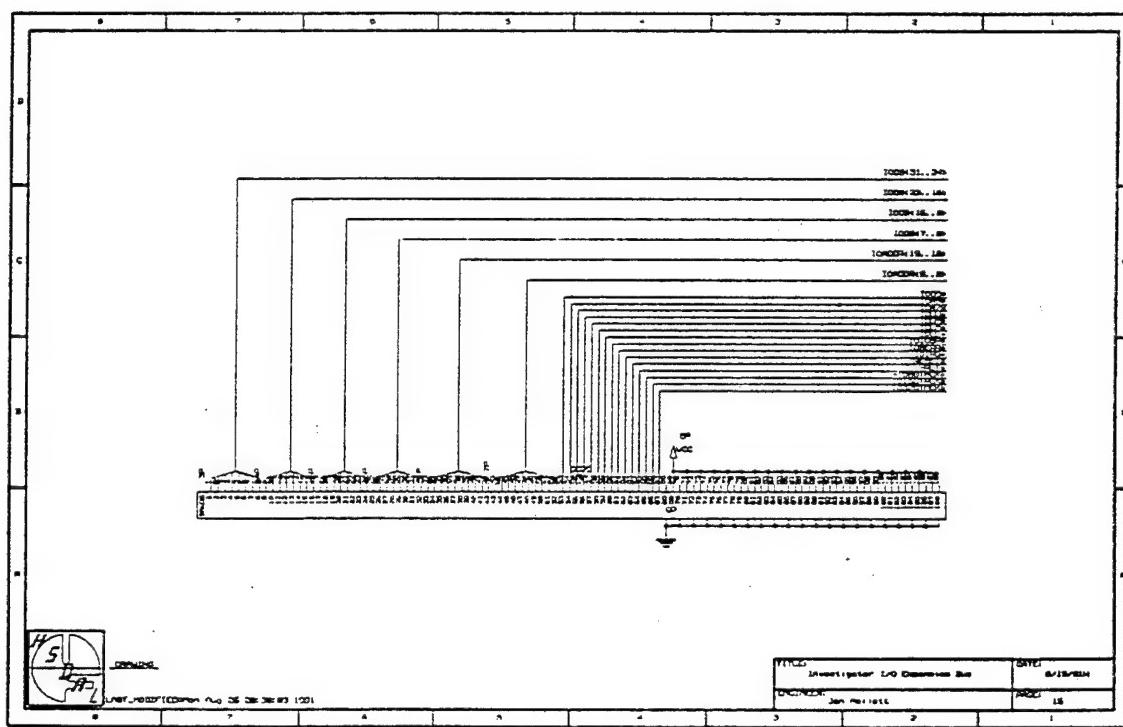
Figure A.12: InvestiGATOR I/O Expansion Bus
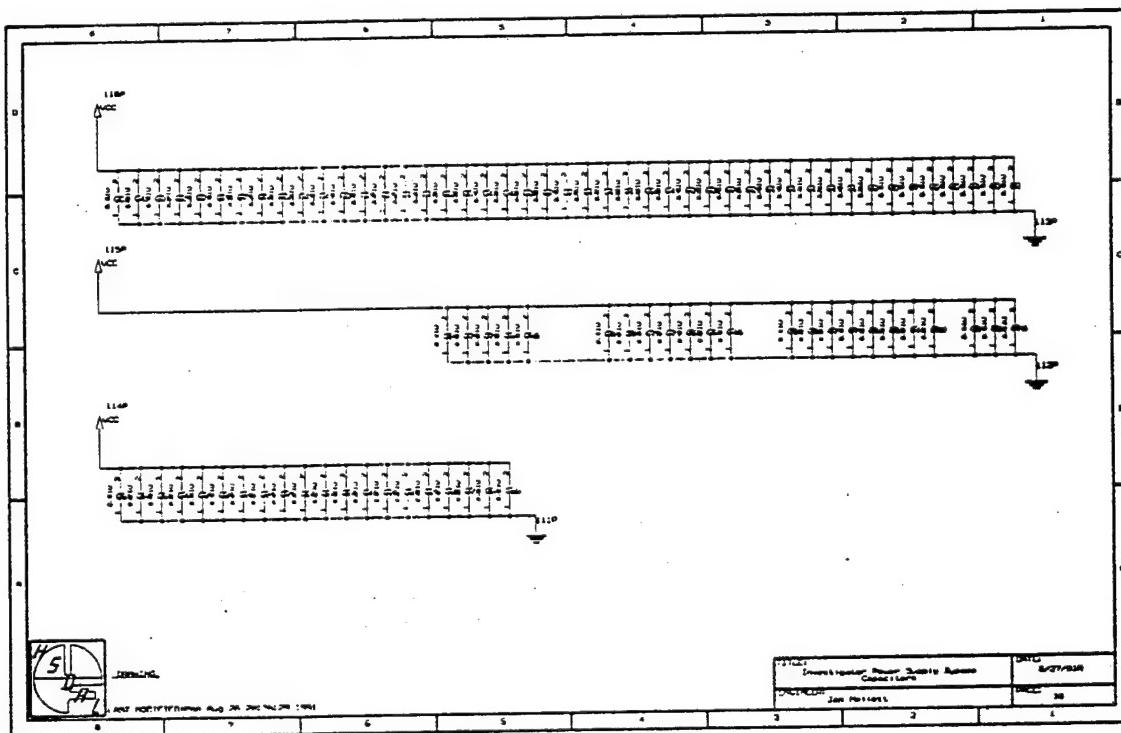
Figure A.13: InvestiGATOR Bypass Capacitors

Appendix B

INVESTIGATOR STATE MACHINES
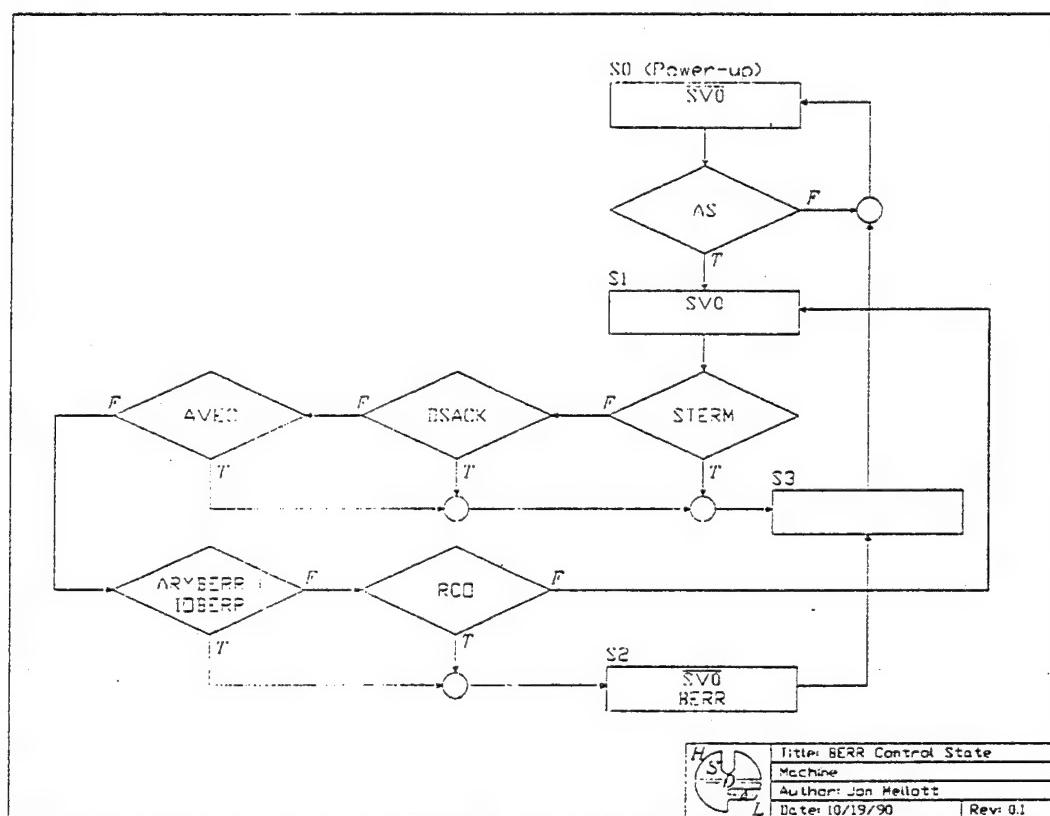
Figure B.1: Bus Error Detection State Machine
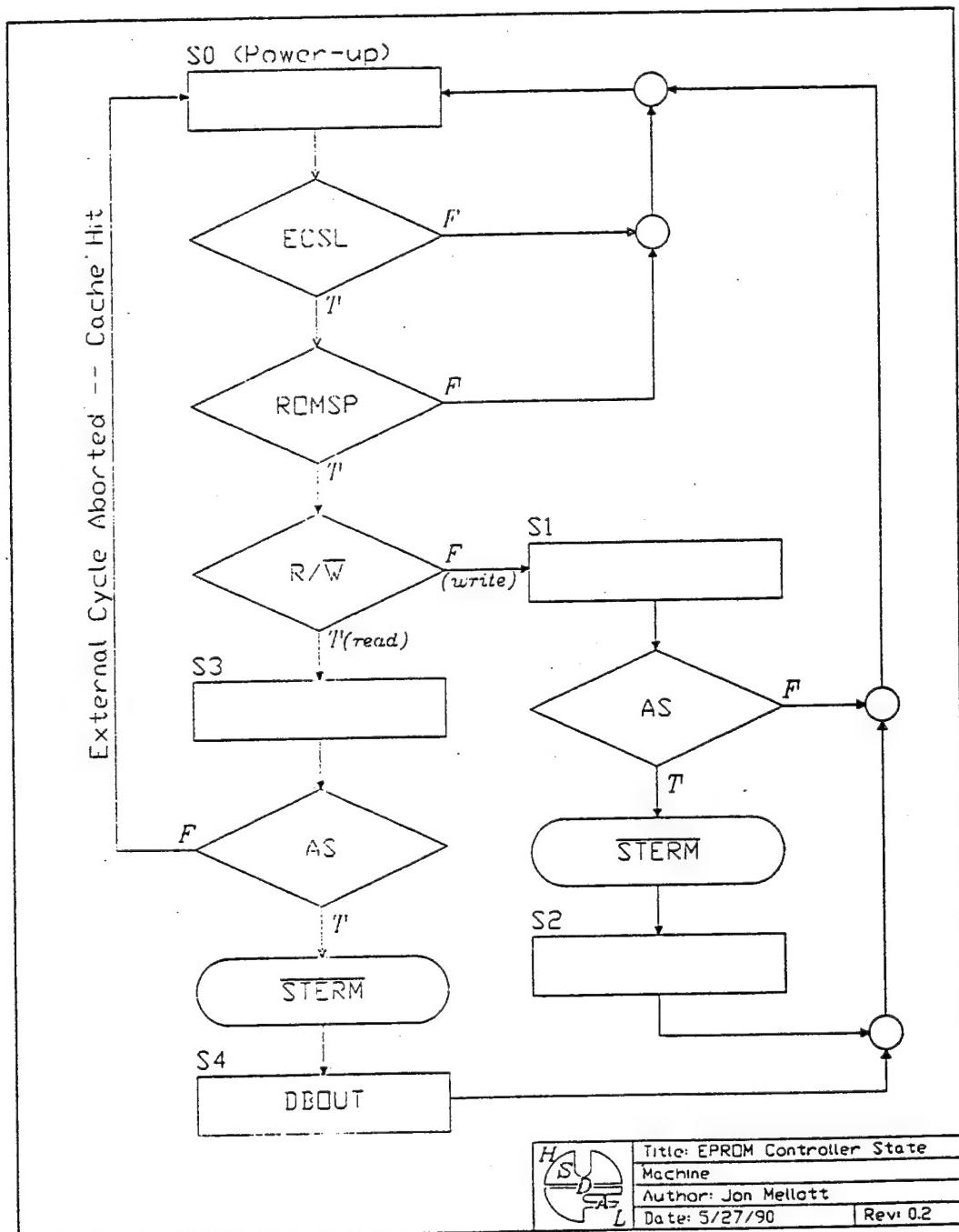
Figure B.2: ROM Controller State Machine

154
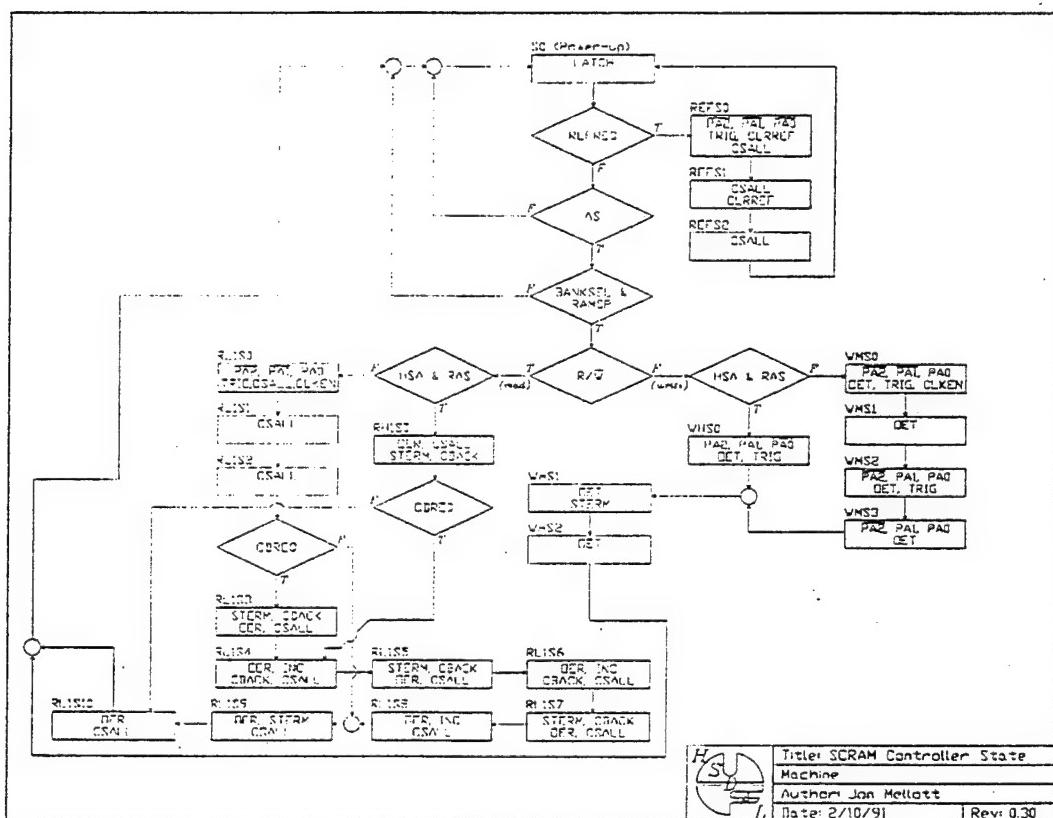
Figure B.3: SCRAM Controller State Machine

155
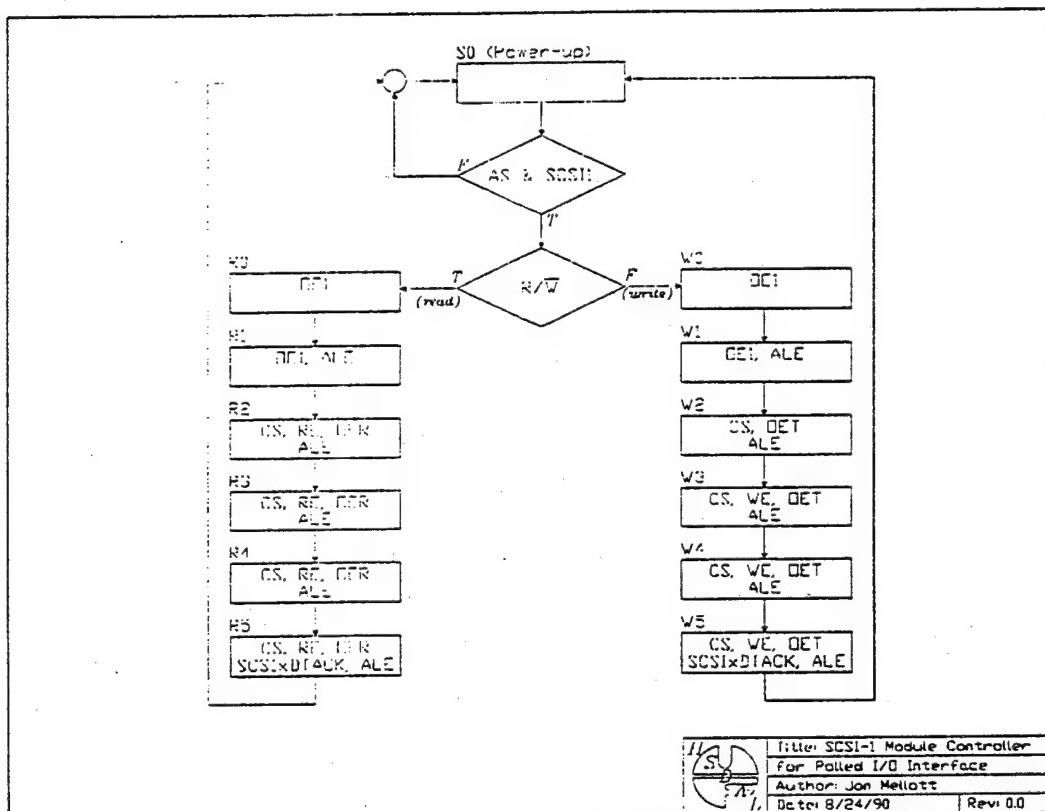
Figure B.1: SBIC Controller State Machine

156
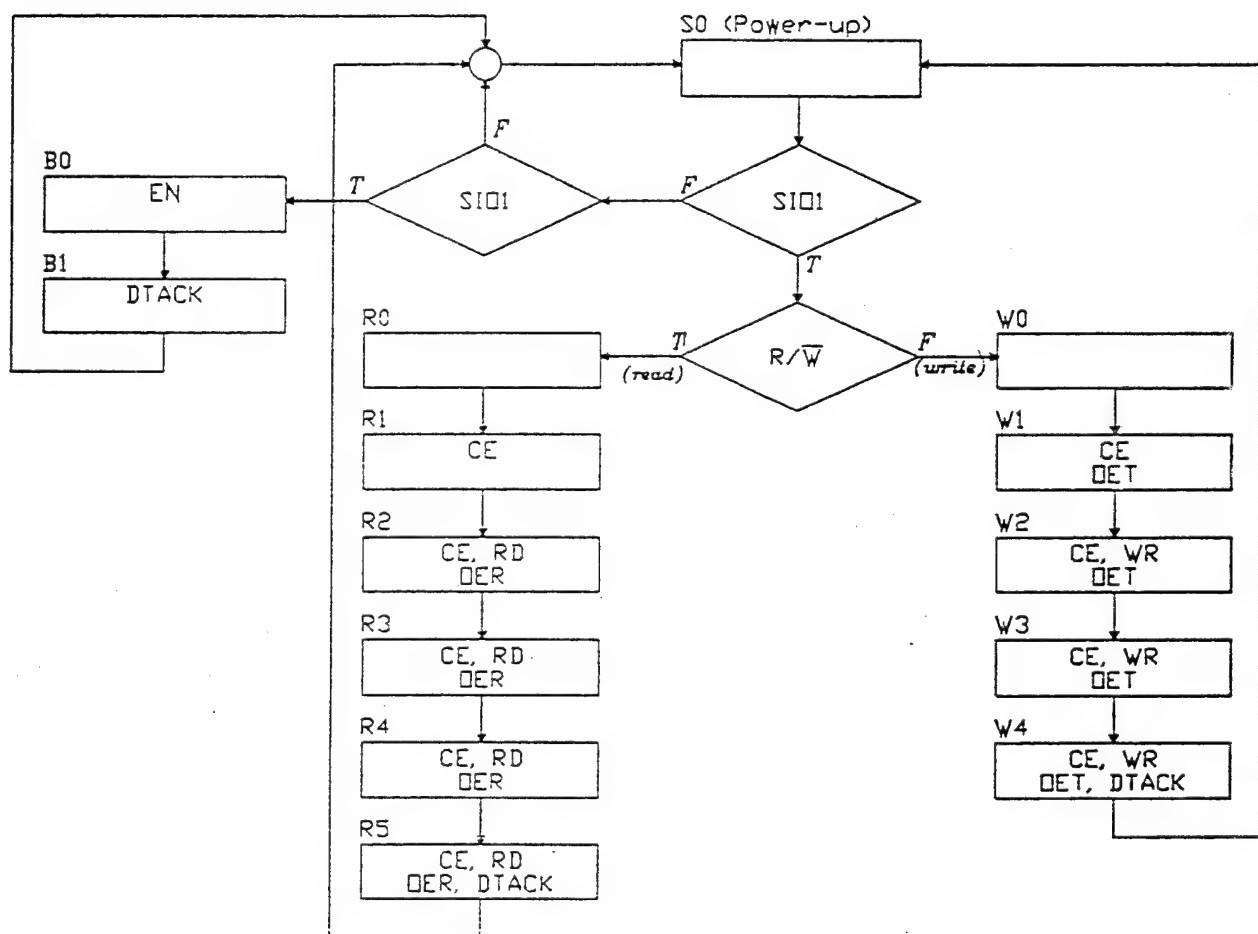
Figure B.5: SIO Controller State Machine

Appendix C

INVESTIGATOR PROGRAMMABLE LOGIC DEVICE LISTINGS

This appendix contains lists for the InvestiGATOR PLDs. These PLDs are used throughout the InvestiGATOR.

C.1 MACH1C

```
TITLE SCSI CONTROLLER STATE MACHINE
PATTERN MACH1
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 2-8-91
CHIP MACH1 MACH210
; PIN/NODE DECLARATIONS
; -------- ------------
;
; P/N  #   NAME              PAIRED WITH PIN     STORAGE         COMMENTS
;----- --- --------------    -------------------  --------------  --------
NODE  1   GLOBAL                                                  ;
PIN   ?   VCC                                                     ; SUPPLY RAIL
PIN   ?   GND                                                     ; GROUND RAIL
PIN   35  CLK20                                  COMBINATORIAL   ; INPUT
PIN   ?   /RESET                                 COMBINATORIAL   ; INPUT
PIN   ?   /SCSI                                  COMBINATORIAL   ; INPUT
PIN   ?   RW                                     COMBINATORIAL   ; INPUT
PIN   ?   /SCSIBUF                               COMBINATORIAL   ; INPUT
PIN   ?   /RCS                                   COMBINATORIAL   ; INPUT
PIN   ?   /DBAS                                  COMBINATORIAL   ; INPUT
PIN   ?   /CS                                    REGISTERED      ; OUTPUT
PIN   ?   ALE                                    REGISTERED      ; OUTPUT
PIN   ?   /RE                                    REGISTERED      ; OUTPUT
PIN   ?   /WE                                    REGISTERED      ; OUTPUT
PIN   ?   /IOR                                   REGISTERED      ; OUTPUT
PIN   ?   /IOT                                   REGISTERED      ; OUTPUT
PIN   ?   /ROE                                   REGISTERED      ; OUTPUT
PIN   ?   /AOE                                   REGISTERED      ; OUTPUT
PIN   ?   DRQ                                    REGISTERED      ; OUTPUT
PIN   ?   /RCE                                   REGISTERED      ; OUTPUT
PIN   ?   /DTACK                                 REGISTERED      ; OUTPUT
PIN   ?   S0                                     COMBINATORIAL   ; OUTPUT
PIN   ?   /ENP                                   REGISTERED      ; OUTPUT
NODE  ?   ENPX                                   REGISTERED      ; OUTPUT
PIN   ?   HIGHZ                                  REGISTERED      ; OUTPUT
NODE  ?   ST0                                    REGISTERED      ; OUTPUT
NODE  ?   ST1                                    REGISTERED      ; OUTPUT
NODE  ?   ST2                                    REGISTERED      ; OUTPUT
```

```
NODE    ?  ST3                          REGISTERED    ; OUTPUT
; PIN    ?  /REIN                       COMBINATORIAL ; TEST INPUT
; PIN    ?  /WEIN                       COMBINATORIAL ; TEST INPUT
;---- --- ------------ --------------------- ------------- --------
; STRING DECLARATIONS
; ------ ------------
; Counter control strings.
STRING CNTLOAD  '/SO'
STRING CNTHOLD  ' SO'
STRING CNTCOUNT ' SO'

STATE
MOORE_MACHINE
CLKF=CLK20
DEFAULT_BRANCH STA0
START_UP := POWER_UP -> STOP

;  STATE TRANSITION EQUATIONS

STOP:= VCC       -> STA0
;  STANDBY FOR RAM OR SBIC ACCESS

STA0:= STANDBY -> STA0
     + DBA      -> STA1
     + SBIC     -> DAMS0
     + RAM      -> RAMS0

;  DBA STANDBY MODE
STA1:= DBA      -> STA1
     + NDBA     -> STA3

;  EXIT DBA STANDBY MODE
STA3 := HOLD    -> STA3
      +-> STA0

;  RAM ACCESS
RAMS0 := READ  -> RAMR1
      + WRITE -> RAMW1

;  RAM READ
RAMR1 := VCC    -> RAMR2
RAMR2 := VCC    -> RAMR3
RAMR3 := VCC    -> STA0

;  RAM WRITE
RAMW1 := VCC    -> RAMW2
RAMW2 := VCC    -> RAMW3
RAMW3 := VCC    -> STA0

;  SBIC ACCESS
DAMS0 := READ   -> DAMR1
       + WRITE -> DAMW1

;  SBIC READ
DAMR1 := VCC    -> DAMR2
DAMR2 := VCC    -> DAMR3
DAMR3 := VCC    -> DAMR4
DAMR4 := VCC    -> DAMR5
DAMR5 := VCC    -> DAMR6
DAMR6 := VCC    -> DAMR7
DAMR7 := VCC    -> STA0

;  SBIC WRITE
DAMW1 := VCC    -> DAMW2
DAMW2 := VCC    -> DAMW3
DAMW3 := VCC    -> DAMW4
DAMW4 := VCC    -> DAMW5
DAMW5 := VCC    -> DAMW6
DAMW6 := VCC    -> DAMW7
DAMW7 := VCC    -> STA0
```

```
;  STATE DEFINITION EQUATIONS
                /ST3*/ST2*/ST1*/ST0*/RE*/WE*/HIGHZ*/CS*/ALE*/IOR*/IOT*/ROE*/DRQ*/RCE*/DTACK*/AOE
STOP  = /ST3*/ST2*/ST1*/ST0*/RE*/WE*/HIGHZ*/CS*/ALE*/IOR*/IOT*/ROE* DRQ* RCE*/DTACK*/AOE
STA0  =  ST3* ST2*/ST1*/ST0*/RE*/WE* HIGHZ*/CS* ALE*/IOR*/IOT*/ROE*/DRQ*/RCE*/DTACK*/AOE
STA1  = /ST3*/ST2*/ST1* ST0*/RE*/WE*/HIGHZ*/CS* ALE*/IOR*/IOT*/ROE* DRQ* RCE*/DTACK*/AOE
STA3  = /ST3*/ST2* ST1*/ST0*/RE*/WE*/HIGHZ*/CS* ALE*/IOR*/IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
RAMS0 = /ST3*/ST2* ST1* ST0*/RE*/WE* HIGHZ*/CS* ALE* IOR*/IOT* ROE* DRQ* RCE*/DTACK*/AOE
RAMR1 = /ST3* ST2*/ST1* ST0*/RE*/WE* HIGHZ*/CS* ALE* IOR*/IOT* ROE* DRQ* RCE* DTACK*/AOE
RAMR2 = /ST3* ST2*/ST1* ST0*/RE*/WE* HIGHZ*/CS* ALE* IOR*/IOT* ROE* DRQ* RCE*/DTACK*/AOE
RAMR3 =  ST3* ST2* ST1*/ST0*/RE*/WE* HIGHZ*/CS* ALE* IOR*/IOT* ROE* DRQ* RCE*/DTACK*/AOE
RAMW1 = /ST3* ST2* ST1* ST0*/RE*/WE* HIGHZ*/CS* ALE*/IOR* IOT*/ROE* DRQ* RCE*/DTACK*/AOE
RAMW2 =  ST3*/ST2*/ST1*/ST0*/RE* WE* HIGHZ*/CS* ALE*/IOR* IOT*/ROE* DRQ* RCE* DTACK*/AOE
RAMW3 =  ST3*/ST2*/ST1* ST0*/RE*/WE* HIGHZ*/CS* ALE*/IOR* IOT*/ROE* DRQ* RCE*/DTACK*/AOE
DAMS0 =  ST3*/ST2* ST1*/ST0*/RE*/WE* HIGHZ*/CS* ALE*/IOR*/IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMR1 =  ST3*/ST2* ST1* ST0*/RE*/WE* HIGHZ*/CS* ALE*/IOR*/IOT*/ROE* DRQ*/RCE*/DTACK* AOE
DAMR2 =  ST3* ST2*/ST1*/ST0*/RE*/WE* HIGHZ*/CS*/ALE*/IOR*/IOT*/ROE* DRQ*/RCE*/DTACK* AOE
DAMR3 =  ST3* ST2*/ST1* ST0* RE*/WE* HIGHZ* CS*/ALE* IOR*/IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMR4 =  ST3* ST2* ST1*/ST0* RE*/WE* HIGHZ* CS*/ALE* IOR*/IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMR5 =  ST3* ST2* ST1* ST0* RE*/WE* HIGHZ* CS*/ALE* IOR*/IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMR6 = /ST3*/ST2*/ST1*/ST0* RE*/WE* HIGHZ* CS*/ALE* IOR*/IOT*/ROE* DRQ*/RCE* DTACK*/AOE
DAMR7 = /ST3*/ST2*/ST1* ST0* RE*/WE* HIGHZ* CS*/ALE* IOR*/IOT*/ROE* DRQ*/RCE*/DTACK* AOE
DAMW1 = /ST3*/ST2* ST1*/ST0*/RE*/WE* HIGHZ*/CS* ALE*/IOR*/IOT*/ROE* DRQ*/RCE*/DTACK* AOE
DAMW2 = /ST3*/ST2* ST1* ST0*/RE*/WE* HIGHZ*/CS*/ALE*/IOR* IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMW3 = /ST3* ST2*/ST1*/ST0*/RE*/WE* HIGHZ* CS*/ALE*/IOR* IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMW4 = /ST3* ST2*/ST1* ST0*/RE* WE* HIGHZ* CS*/ALE*/IOR* IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMW5 = /ST3* ST2* ST1*/ST0*/RE* WE* HIGHZ* CS*/ALE*/IOR* IOT*/ROE* DRQ*/RCE*/DTACK*/AOE
DAMW6 = /ST3* ST2* ST1* ST0*/RE* WE* HIGHZ* CS*/ALE*/IOR* IOT*/ROE* DRQ*/RCE* DTACK*/AOE
DAMW7 =  ST3*/ST2*/ST1*/ST0*/RE* WE* HIGHZ* CS*/ALE*/IOR* IOT*/ROE* DRQ*/RCE*/DTACK*/AOE


;  OUTPUT EQUATIONS are now removed
STOP.OUTF  =CNTHOLD
STA0.OUTF  =CNTHOLD
STA1.OUTF  =CNTCOUNT
STA3.OUTF  =CNTCOUNT
RAMS0.OUTF =CNTHOLD
RAMR1.OUTF =CNTLOAD
RAMR2.OUTF =CNTHOLD
RAMR3.OUTF =CNTHOLD
RAMW1.OUTF =CNTLOAD
RAMW2.OUTF =CNTHOLD
RAMW3.OUTF =CNTHOLD
DAMS0.OUTF =CNTHOLD
DAMR1.OUTF =CNTHOLD
DAMR2.OUTF =CNTHOLD
DAMR3.OUTF =CNTHOLD
DAMR4.OUTF =CNTHOLD
DAMR5.OUTF =CNTHOLD
DAMR6.OUTF =CNTHOLD
DAMR7.OUTF =CNTHOLD
DAMW1.OUTF =CNTHOLD
DAMW2.OUTF =CNTHOLD
DAMW3.OUTF =CNTHOLD
DAMW4.OUTF =CNTHOLD
DAMW5.OUTF =CNTHOLD
DAMW6.OUTF =CNTHOLD
DAMW7.OUTF =CNTHOLD

CONDITIONS

;  Switch to DBA mode
DBA = DBAS

;  Switch from DBA mode to standby mode
NDBA = /DBAS
```

```
; SBIC access
SBIC = SCSI * /DBAS

; RAM access
RAM = SCSIBUF * /SCSI * /DBAS

; No access
STANDBY = /DBAS * /SCSI * /SCSIBUF

; Hold while exiting DBA mode
HOLD = /RCS

; Read cycle
READ = RW

; Write cycle
WRITE = /RW

EQUATIONS

GLOBAL.RSTF = RESET

; Tri-state control for RE and WE strobes.
WE.TRST = HIGHZ
RE.TRST = HIGHZ

; Counter control for DBA mode accesses.
ENP   := /ENP *  ENPX * /(RCS*(RE+WE))
ENPX  := /ENP * /ENPX *  (RCS*(RE+WE)) + /ENP * ENPX
ENP.CLKF  = CLK20
ENPX.CLKF = CLK20

; Counter control version for simulation
; ENP   := /ENP *  ENPX * /(RCS*(REIN+WEIN))
; ENPX  := /ENP * /ENPX *  (RCS*(REIN+WEIN)) + /ENP * ENPX

SIMULATION

TRACE_ON CLK20 RESET SCSI RW SCSIBUF DBAS CS ALE RE WE IOR IOT ROE AOE DRQ
 RCE DTACK SO ENP HIGHZ

SETF /CLK20 /DBAS /SCSI /SCSIBUF /RESET /RW /RCS
; SETF RESET
; CLOCKF ;STOP
; SETF /RESET
CLOCKF ;STA0
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE
CLOCKF ;STA0
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE

; RAM READ
SETF SCSIBUF RW
CLOCKF ; RAMSO
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ /RCE /DTACK /AOE
CLOCKF ; RAMR1
CHECK /ENP /SO
CHECK /RE /WE HIGHZ /CS ALE  IOR /IOT  ROE DRQ RCE /DTACK /AOE
CLOCKF ;RAMR2
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE  IOR /IOT  ROE DRQ RCE  DTACK /AOE
CLOCKF ;RAMR3
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE  IOR /IOT  ROE DRQ RCE /DTACK /AOE
CLOCKF ;STA0
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE
```

```
; RAM WRITE
SETF SCSIBUF /RW
CLOCKF ; RAMS0
CHECK /ENP SO
CHECK /RE /WE  HIGHZ /CS ALE /IOR /IOT /ROE DRQ /RCE /DTACK /AOE
CLOCKF ;RAMW1
CHECK /ENP /SO
CHECK /RE /WE HIGHZ /CS ALE /IOR  IOT /ROE DRQ RCE /DTACK /AOE
CLOCKF ;RAMW2
CHECK /ENP SO
CHECK /RE  WE HIGHZ /CS ALE /IOR  IOT /ROE DRQ RCE  DTACK /AOE
CLOCKF ;RAMW3  ·
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR  IOT /ROE DRQ RCE /DTACK /AOE
CLOCKF ;STA0
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE
SETF /DBAS /SCSI /SCSIBUF

; DBA standby mode
SETF DBAS
CLOCKF ; STA1
CHECK /ENP SO
CHECK /HIGHZ /CS  ALE /IOR /IOT /ROE /DRQ  RCE /DTACK /AOE
CLOCKF ; STA1
CHECK /ENP SO
CHECK /HIGHZ /CS  ALE /IOR /IOT /ROE /DRQ  RCE /DTACK /AOE
SETF RCS
; SETF RCS /REIN /WEIN; BEGIN DBA TRANSACTIONS
; CLOCKF ; STA1
; CHECK /ENP SO /ENPX
; SETF RCS WEIN
; CLOCKF ; STA1
; CHECK /ENP SO  ENPX
; CLOCKF ; STA1
; CHECK /ENP SO  ENPX
; SETF RCS /WEIN
; CLOCKF ; STA1
; CHECK  ENP SO  ENPX
; CLOCKF ; STA1
; CHECK /ENP SO /ENPX
; SETF RCS REIN
; CLOCKF ; STA1
; CHECK /ENP SO  ENPX
; SETF RCS /REIN
; CLOCKF ; STA1
; CHECK  ENP SO  ENPX
; CLOCKF ; STA1
; CHECK /ENP SO /ENPX

CLOCKF ;STA1
CHECK /ENP SO
CHECK /HIGHZ /CS  ALE /IOR /IOT /ROE /DRQ  RCE /DTACK /AOE
SETF /DBAS
CLOCKF ;STA3
CHECK /ENP SO
CHECK /HIGHZ /CS  ALE /IOR /IOT /ROE DRQ  RCE /DTACK /AOE
CLOCKF ;STA0
CHECK /ENP SO
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE
SETF /RCS
; SBIC READ
```

```
SETF SCSI RW
CLOCKF ; DAMS0
CHECK /ENP S0
CHECK /RE /WE   HIGHZ /CS  ALE /IOR /IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMR1
CHECK /ENP S0
CHECK /RE /WE   HIGHZ /CS  ALE /IOR /IOT /ROE  DRQ /RCE /DTACK  AOE
CLOCKF ; DAMR2
CHECK /ENP S0
CHECK /RE /WE   HIGHZ /CS /ALE /IOR /IOT /ROE  DRQ /RCE /DTACK  AOE
CLOCKF ; DAMR3
CHECK /ENP S0
CHECK  RE /WE   HIGHZ  CS /ALE  IOR /IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMR4
CHECK /ENP S0
CHECK  RE /WE   HIGHZ  CS /ALE  IOR /IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMR5
CHECK /ENP S0
CHECK  RE /WE   HIGHZ  CS /ALE  IOR /IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMR6
CHECK /ENP S0
CHECK  RE /WE   HIGHZ  CS /ALE  IOR /IOT /ROE  DRQ /RCE  DTACK /AOE
CLOCKF ; DAMR7
CHECK /ENP S0
CHECK  RE /WE   HIGHZ  CS /ALE  IOR /IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ;STA0
CHECK /ENP S0
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE

; SBIC WRITE
SETF SCSI /RW
CLOCKF ; DAMS0
CHECK /ENP S0
CHECK /RE /WE   HIGHZ /CS  ALE /IOR /IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMW1
CHECK /ENP S0
CHECK /RE /WE   HIGHZ /CS  ALE /IOR /IOT /ROE  DRQ /RCE /DTACK. AOE
CLOCKF ; DAMW2
CHECK /ENP S0
CHECK /RE /WE   HIGHZ /CS /ALE /IOR /IOT /ROE  DRQ /RCE /DTACK  AOE
CLOCKF ; DAMW3
CHECK /ENP S0
CHECK /RE /WE   HIGHZ  CS /ALE /IOR  IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMW4
CHECK /ENP S0
CHECK /RE  WE   HIGHZ  CS /ALE /IOR  IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMW5
CHECK /ENP S0
CHECK /RE  WE   HIGHZ  CS /ALE /IOR  IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ; DAMW6
CHECK /ENP S0
CHECK /RE  WE   HIGHZ  CS /ALE /IOR  IOT /ROE  DRQ /RCE  DTACK /AOE
CLOCKF ; DAMW7
CHECK /ENP S0
CHECK /RE  WE   HIGHZ  CS /ALE /IOR  IOT /ROE  DRQ /RCE /DTACK /AOE
CLOCKF ;STA0
CHECK /ENP S0
CHECK /RE /WE HIGHZ /CS ALE /IOR /IOT /ROE DRQ RCE /DTACK /AOE
TRACE_OFF
```

C.2 MACH2

```
TITLE CPU MODULE SUPPORT GLUE
PATTERN MACH2
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 2-9-91
;    MACH2 also provides interrupt request encoding for the 030's IPL
;    lines.  Independent interrupt request lines are provided for
;    both SCSI interfaces, the serial I/O interface, the timer,
;    any additional devices on the I/O bus, and the array bus.
;    The interrupts are prioritized as follows:
;
;    Interrupt
;    Request
;    Priority     Description
;    ---------    -----------
;       7         NMI.  Reserved.
;       6         SCSI1.
;       5         Unallocated IRQ.  Reserved.
;       4         SIO.
;       3         Timer.
;       2         I/O Bus.
;       1         Array Bus.
;
CHIP MACH2 MACH110

; PIN/NODE DECLARATIONS
; -------- ------------
;
; P/N  #   NAME            PAIRED WITH PIN      STORAGE         COMMENTS
;----  --- -----------     -----------------    -------------   --------
NODE   1   GLOBAL
PIN    ?   VCC                                                  ; SUPPLY RAIL
PIN    ?   GND                                                  ; GROUND RAIL

PIN   35   CLK20                                COMBINATORIAL   ; INPUT

PIN    ?   /RESET                               COMBINATORIAL   ; INPUT
PIN    ?   /AS                                  COMBINATORIAL   ; INPUT
PIN    ?   /AVEC                                COMBINATORIAL   ; INPUT
PIN    ?   /DSACK0                              COMBINATORIAL   ; INPUT
PIN    ?   /DSACK1                              COMBINATORIAL   ; INPUT
PIN    ?   /STERM                               COMBINATORIAL   ; INPUT
PIN    ?   /ARYBERR                             COMBINATORIAL   ; INPUT
PIN    ?   /IOBERR                              COMBINATORIAL   ; INPUT
PIN    ?   /RCO                                 COMBINATORIAL   ; INPUT
PIN    ?   /BERR                                REGISTERED      ; OUTPUT
PIN    ?   SV0                                  REGISTERED      ; OUTPUT
NODE   ?   SV1                                  REGISTERED      ; OUTPUT

PIN    ?   SCSI1IRQ                             COMBINATORIAL   ; INPUT
PIN    ?   /SIOIRQ                              COMBINATORIAL   ; INPUT
PIN    ?   /TIMERIRQ                            COMBINATORIAL   ; INPUT
PIN    ?   /IOIRQ                               COMBINATORIAL   ; INPUT
PIN    ?   /ARYIRQ                              COMBINATORIAL   ; INPUT
PIN    ?   /IPL2                                COMBINATORIAL   ; OUTPUT
PIN    ?   /IPL1                                COMBINATORIAL   ; OUTPUT
PIN    ?   /IPL0                                COMBINATORIAL   ; OUTPUT
;----  --- -----------     -----------------    -------------   --------

; STRING DECLARATIONS
; ------ ------------
```

```
STATE
MOORE_MACHINE
CLKF=CLK20
DEFAULT_BRANCH S0
START_UP := POWER_UP -> S0

;   STATE DEFINITIONS
; Power-up & standby.
S0 = /SV1 */SV0 * /BERR

; Cycle in progress.
S1 = /SV1 *SV0 * /BERR

; Bus error detected.
S2 = /SV1 * /SV0 *  BERR
S3 =  SV1 * /SV0 * /BERR

;   STATE TRANSITION EQUATIONS
;   WAIT FOR CYCLE TO START STATE
S0 := WAIT      -> S0
   + GO         -> S1
;   WAIT FOR NORMAL CYCLE COMPLETION STATE
S1 := STANDBY   -> S1
   + DONE_OK   -> S3
   + FAULT     -> S2
;   BERR HANDLER STATE
S2 := VCC       -> S3
;   RETURN TO S0
S3 := VCC       -> S0

; OUTPUT EQUATIONS

CONDITIONS

;   HOLD AT S1
WAIT    = /AS
;   START CYCLE
GO      = AS
;   STANDBY FOR COMPLETION OF CYCLE
STANDBY = /AVEC * /DSACK0 * /DSACK1 * /STERM *
   /ARYBERR * /IOBERR * /RCO
;   END CYCLE
DONE_OK = (AVEC + DSACK0 + DSACK1 + STERM) *
   /(ARYBERR + IOBERR + RCO)
;   BUS FAULT
FAULT   = /AVEC * /DSACK0 * /DSACK1 * /STERM *
   (ARYBERR + IOBERR + RCO)

EQUATIONS

;   Interrupt priority encoding.
IPL2 = SCSI1IRQ + SIOIRQ
IPL1 = SCSI1IRQ + (TIMERIRQ + IOIRQ)*/SIOIRQ
IPL0 = TIMERIRQ * /(SIOIRQ+SCSI1IRQ) + ARYIRQ * /(IOIRQ+SIOIRQ+SCSI1IRQ)

GLOBAL.RSTF = RESET
; GLOBAL.CLKF = CLK20

SIMULATION

TRACE_ON BERR SV0 SV1 SCSI1IRQ SIOIRQ TIMERIRQ IOIRQ ARYIRQ IPL2 IPL1 IPL0

SETF /CLK20 RESET /AS /AVEC /DSACK0 /DSACK1 /STERM /ARYBERR /IOBERR /RCO
CLOCKF
```

```
    SETF /RESET
    ; *EXHAUSTIVELY* TEST IPL ENCODING...
    SETF /SCSI1IRQ /SIOIRQ /TIMERIRQ /IOIRQ /ARYIRQ
    CHECK /IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ /SIOIRQ /TIMERIRQ /IOIRQ  ARYIRQ
    CHECK /IPL2 /IPL1  IPL0
    SETF /SCSI1IRQ /SIOIRQ /TIMERIRQ  IOIRQ /ARYIRQ
    CHECK /IPL2  IPL1 /IPL0
    SETF /SCSI1IRQ /SIOIRQ /TIMERIRQ  IOIRQ  ARYIRQ
    CHECK /IPL2  IPL1 /IPL0
    SETF /SCSI1IRQ /SIOIRQ  TIMERIRQ /IOIRQ /ARYIRQ
    CHECK /IPL2  IPL1  IPL0
    SETF /SCSI1IRQ /SIOIRQ  TIMERIRQ /IOIRQ  ARYIRQ
    CHECK /IPL2  IPL1  IPL0
    SETF /SCSI1IRQ /SIOIRQ  TIMERIRQ  IOIRQ /ARYIRQ
    CHECK /IPL2  IPL1  IPL0
    SETF /SCSI1IRQ /SIOIRQ  TIMERIRQ  IOIRQ  ARYIRQ
    CHECK /IPL2  IPL1  IPL0
    SETF /SCSI1IRQ  SIOIRQ /TIMERIRQ /IOIRQ /ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ /TIMERIRQ /IOIRQ  ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ /TIMERIRQ  IOIRQ /ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ /TIMERIRQ  IOIRQ  ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ  TIMERIRQ /IOIRQ /ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ  TIMERIRQ /IOIRQ  ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ  TIMERIRQ  IOIRQ /ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF /SCSI1IRQ  SIOIRQ  TIMERIRQ  IOIRQ  ARYIRQ
    CHECK  IPL2 /IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ /TIMERIRQ /IOIRQ /ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ /TIMERIRQ /IOIRQ  ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ /TIMERIRQ  IOIRQ /ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ /TIMERIRQ  IOIRQ  ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ  TIMERIRQ /IOIRQ /ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ  TIMERIRQ /IOIRQ  ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ  TIMERIRQ  IOIRQ /ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ /SIOIRQ  TIMERIRQ  IOIRQ  ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ  SIOIRQ /TIMERIRQ /IOIRQ /ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ  SIOIRQ /TIMERIRQ /IOIRQ  ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ  SIOIRQ /TIMERIRQ  IOIRQ /ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ  SIOIRQ /TIMERIRQ  IOIRQ  ARYIRQ
    CHECK  IPL2  IPL1 /IPL0
    SETF  SCSI1IRQ  SIOIRQ  TIMERIRQ /IOIRQ /ARYIRQ
```

```
    :

CHECK  IPL2  IPL1 /IPLO
SETF  SCSI1IRQ  SIOIRQ  TIMERIRQ /IOIRQ  ARYIRQ
CHECK  IPL2  IPL1 /IPLO
SETF  SCSI1IRQ  SIOIRQ  TIMERIRQ  IOIRQ /ARYIRQ
CHECK  IPL2  IPL1 /IPLO
SETF  SCSI1IRQ  SIOIRQ  TIMERIRQ  IOIRQ  ARYIRQ
CHECK  IPL2  IPL1 /IPLO

; TEST BERR STATE MACHINE
SETF RESET
CLOCKF ; SO
SETF /RESET /STERM /DSACK1 /DSACKO /AVEC /ARYBERR /IOBERR /RCO
CHECK /SV1 /SVO /BERR
CLOCKF ;SO
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
SETF STERM
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /STERM
CLOCKF ;SO
CHECK /SV1 /SVO /BERR
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
SETF DSACKO
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /DSACKO
CLOCKF ;SO
CHECK /SV1 /SVO /BERR
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
SETF DSACK1
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /DSACK1
CLOCKF ;SO
CHECK /SV1 /SVO /BERR
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
SETF AVEC
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /AVEC
CLOCKF ;SO
CHECK /SV1 /SVO /BERR
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
```

```
SETF ARYBERR
CLOCKF ;S2
CHECK /SV1 /SVO BERR
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /ARYBERR
CLOCKF ;S0
CHECK /SV1 /SVO /BERR
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
SETF IOBERR
CLOCKF ;S2
CHECK /SV1 /SVO BERR
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /IOBERR
CLOCKF ;S0
CHECK /SV1 /SVO /BERR
SETF AS
CLOCKF ;S1
CHECK /SV1 SVO /BERR
CLOCKF ;S1
CHECK /SV1 SVO /BERR
SETF RCO
CLOCKF ;S2
CHECK /SV1 /SVO BERR
CLOCKF ;S3
CHECK  SV1 /SVO /BERR
SETF /AS /RCO
CLOCKF ;S0
CHECK /SV1 /SVO /BERR

TRACE_OFF
```

C.3 MACH3XA

```
TITLE SCRAM CONTROLLER
PATTERN MACH3XA
REVISION 0.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 2-2-92

CHIP MACH3 MACH210

; PIN/NODE DECLARATIONS
; -------- ------------
;
; P/N  #  NAME          PAIRED WITH PIN       STORAGE         COMMENTS
;----  ---  -----------  --------------------  --------------  --------
NODE   1  GLOBAL

PIN   35  CLK20                                COMBINATORIAL ; INPUT

PIN   10  /RESET                               COMBINATORIAL ; INPUT
PIN   33  /AS                                  COMBINATORIAL ; INPUT
PIN   13  /RAMSP                               COMBINATORIAL ; INPUT
PIN   32  RW                                   COMBINATORIAL ; INPUT
PIN   25  /CBREQ                               COMBINATORIAL ; INPUT
PIN   11  /BANKSEL                             COMBINATORIAL ; INPUT
```

```
PIN   27   /RCO                        COMBINATORIAL  ; INPUT

PIN   31   /HSA                        REGISTERED     ; OUTPUT
PIN   30   /RAS                        REGISTERED     ; OUTPUT
PIN    4   /CSALL                      REGISTERED     ; OUTPUT
PIN    5   /WE                         REGISTERED     ; OUTPUT
PIN    6   /CS                         REGISTERED     ; OUTPUT
PIN   21   /OE                         REGISTERED     ; OUTPUT
PIN    8   /STERM                      REGISTERED     ; OUTPUT
PIN   14   /CBACK                      REGISTERED     ; OUTPUT
PIN    2   TRIGJ                       REGISTERED     ; OUTPUT
PIN    3   TRIGK                       REGISTERED     ; OUTPUT
PIN    7   /OER                        REGISTERED     ; OUTPUT
PIN   24   /OET                        REGISTERED     ; OUTPUT
PIN    9   CLKEN                       REGISTERED     ; OUTPUT
NODE   ?   LTCH                        REGISTERED     ; OUTPUT
NODE   ?   INC                         REGISTERED     ; OUTPUT
PIN   41   /CLRREF                     REGISTERED     ; OUTPUT
PIN   43   /REFREQ                     REGISTERED     ; OUTPUT
NODE   ?   SV1                         REGISTERED     ; OUTPUT
NODE   ?   SV0                         REGISTERED     ; OUTPUT

PIN   40   AO0                         REGISTERED     ; OUTPUT
PIN   42   AO1                         REGISTERED     ; OUTPUT
PIN   28   AI0                         COMBINATORIAL  ; INPUT
PIN   29   AI1                         COMBINATORIAL  ; INPUT
;----  ---  -------------   -------------------- --------------  --------

; The PAx outputs which would normally be used to trigger the PEG
; are used for write strobe (WE*), chip select (CS*), and output
; enable (OE*) in this implementation. Additionally, the RAS input
; becomes the RAS output. Note, with the four aforementioned signals,
; it is assumed that they are distributed at the PEG socket.

; STRING DECLARATIONS
; ------ ------------
STATE
MOORE_MACHINE
CLKF=CLK20
DEFAULT_BRANCH S0
START_UP := POWER_UP -> STOP

;  STATE TRANSITION EQUATIONS
;  POWER-UP STATE
STOP := VCC      -> S0

;  HOLD STATE
S0  := WAIT      -> S0
    + REF_REQ  -> REFS0
    + READ     -> SR1
    + WRITE    -> SW0

;  REFRESH STATE SEQUENCE
REFS0 := VCC -> REFS1
REFS1 := VCC -> S0

;  NON-STATIC COLUMN READ SEQUENCE
SR1    := BRST_REQ -> SBR1
       + NO_B_REQ -> SR2
SR2    := VCC        -> SBR7
SBR7   := VCC        -> SR3
SR3    := VCC        -> S0

;  STATIC COLUMN READ SEQUENCE
SBR1   := VCC -> SBR2
SBR2   := VCC -> SBR3
SBR3   := VCC -> SBR4
```

```
SBR4   := VCC -> SBR5
SBR5   := VCC -> SBR6
SBR6   := VCC -> SBR7
;  NON-STATIC COLUMN WRITE SEQUENCE
SW0    := VCC -> SW1
SW1    := VCC -> SW2
SW2    := VCC -> S0
; STATE DEFINITION EQUATIONS
STOP  = /SV1*/SV0*/LTCH*/CSALL*/STERM*/CBACK*/OER*/OET*/INC*/CLRREF*/RAS*/WE*/CS*/OE
S0    = /SV1*/SV0* LTCH*/CSALL*/STERM*/CBACK*/OER*/OET*/INC*/CLRREF*/RAS*/WE*/CS*/OE
REFS0 = /SV1*/SV0*/LTCH* CSALL*/STERM*/CBACK*/OER*/OET*/INC* CLRREF*/RAS*/WE* CS*/OE
REFS1 = /SV1*/SV0*/LTCH* CSALL*/STERM*/CBACK*/OER*/OET*/INC*/CLRREF* RAS*/WE* CS*/OE
SR1   = /SV1*/SV0*/LTCH* CSALL*/STERM*/CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE*/CS* OE
SBR1  = /SV1*/SV0*/LTCH* CSALL* STERM* CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE* CS* OE
SBR2  = /SV1*/SV0*/LTCH* CSALL*/STERM* CBACK* OER*/OET* INC*/CLRREF* RAS*/WE* CS* OE
SBR3  = /SV1*/SV0*/LTCH* CSALL* STERM* CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE* CS* OE
SBR4  = /SV1* SV0*/LTCH* CSALL*/STERM* CBACK* OER*/OET* INC*/CLRREF* RAS*/WE* CS* OE
SBR5  = /SV1* SV0*/LTCH* CSALL* STERM* CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE* CS* OE
SBR6  = /SV1*/SV0*/LTCH* CSALL* STERM*/CBACK* OER*/OET* INC*/CLRREF* RAS*/WE* CS* OE
SBR7  = /SV1*/SV0*/LTCH* CSALL* STERM*/CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE* CS* OE
SR2   = /SV1*/SV0*/LTCH* CSALL*/STERM*/CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE* CS* OE
SR3   = /SV1* SV0*/LTCH* CSALL*/STERM*/CBACK* OER*/OET*/INC*/CLRREF* RAS*/WE* CS* OE
SW0   = /SV1*/SV0*/LTCH*/CSALL*/STERM*/CBACK*/OER* OET*/INC*/CLRREF* RAS* WE*/CS*/OE
SW1   = /SV1*/SV0*/LTCH*/CSALL*/STERM*/CBACK*/OER* OET*/INC*/CLRREF* RAS* WE* CS*/OE

; OUTPUT EQUATIONS
;STOP.OUTF   = /PA2*/PA1*/PA0
;S0.OUTF     = /PA2*/PA1*/PA0
;RL1S0.OUTF  =  PA2*/PA1*/PA0
;RL1S1.OUTF  =  PA2*/PA1*/PA0
;RL1S2.OUTF  = /PA2*/PA1*/PA0
;RL1S3.OUTF  = /PA2*/PA1*/PA0
;RL1S4.OUTF  = /PA2*/PA1*/PA0
;RL1S5.OUTF  = /PA2*/PA1*/PA0
;RL1S6.OUTF  = /PA2*/PA1*/PA0
;RL1S7.OUTF  = /PA2*/PA1*/PA0
;RL1S8.OUTF  = /PA2*/PA1*/PA0
;RL1S9.OUTF  = /PA2*/PA1*/PA0
;RL1S10.OUTF = /PA2*/PA1*/PA0
;RH1S0.OUTF  = /PA2*/PA1*/PA0
;WHS0.OUTF   = /PA2* PA1*/PA0
;WHS1.OUTF   = /PA2* PA1*/PA0
;WHS2.OUTF   = /PA2*/PA1*/PA0
;WMS0.OUTF   = /PA2*/PA1* PA0
;WMS1.OUTF   = /PA2*/PA1* PA0
;WMS2.OUTF   = /PA2* PA1* PA0
;WMS3.OUTF   = /PA2* PA1* PA0
;REFS0.OUTF  = /PA2*/PA1*/PA0
;REFS1.OUTF  = /PA2*/PA1*/PA0
;REFS2.OUTF  = /PA2*/PA1*/PA0

CONDITIONS
;  HOLD AT S0
WAIT    = (/REFREQ * /AS * /(RAMSP * BANKSEL)) + (/REFREQ * RESET)
;  REFRESH REQUEST
REF_REQ = REFREQ
;  READ REQUEST
READ    = /REFREQ * AS * (BANKSEL * RAMSP) * RW * /RESET
```

```
;   WRITE REQUEST
WRITE   = /REFREQ * AS * (BANKSEL * RAMSP) * /RW * /RESET

;   BURST REQUEST
BRST_REQ = CBREQ
NO_B_REQ = /CBREQ

EQUATIONS

GLOBAL.RSTF = RESET
; GLOBAL.CLKF = CLK20

REFREQ := RCO*/CLRREF + REFREQ*/CLRREF
REFREQ.CLKF = CLK20

; BURST ADDRESS COUNTER EQUATIONS

A00 := LTCH*AI0 + (/LTCH*/INC)*A00 + (/LTCH*INC)*/A00
A00.CLKF = CLK20
A00.TRST = VCC

A01 := LTCH*AI1 + (/LTCH*/INC)*A01 + (/LTCH*INC)*(/A01*A00+A01*/A00)
A01.CLKF = CLK20
A01.TRST = VCC

; UNUSED SIGNALS IN THIS VERSION
CLKEN     := GND
CLKEN.CLKF = CLK20
TRIGJ     := GND
TRIGJ.CLKF = CLK20
TRIGK     := GND
TRIGK.CLKF = CLK20
HSA       :=GND
HSA.CLKF = CLK20
HSA.TRST = VCC

SIMULATION
```

C.4 PAL0

```
TITLE ADDRESS SPACE DECODER
PATTERN PAL0
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 5-20-90

;    PAL0 MUST BE PAL22V10-10 DC
;    -- PAL22V10-10 -- 10 ns
;    -- D:  ceramic package
;    -- C:  Commercial grade

;    DESCRIPTION:
;    This PAL decodes the address lines into the appropriate address space.
;    The memory map is as follows:
;
;    |  repeats      |
;    +---------------+ 64Mb
;    |               |
;    |               |
;    | Array         |
;    | Space         |
;    |               |
;    |               |
;    +---------------+ 16Mb
```
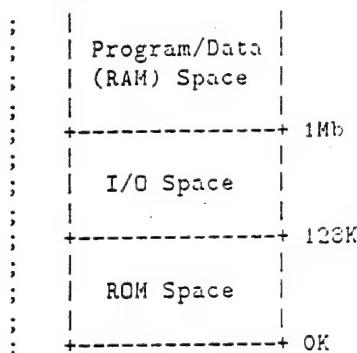
```
;      |          |
;      | Program/Data |
;      | (RAM) Space  |
;      |          |
;      +--------------+ 1Mb
;      |          |
;      | I/O Space    |
;      |          |
;      +--------------+ 128K
;      |          |
;      | ROM Space    |
;      |          |
;      +--------------+ OK

;    The outputs are conditioned by the /RESET signal so that the outputs
;    are held false during reset.  Additionally, the outputs are conditioned
;    by the function code signal so that memory does not respond during
;    CPU space cycles.  CPU space is indicated by FC0=FC1=FC2=1.

;    The cache inhibit input is generated by this PAL.
;    Caching is inhibited in I/O space and Array space.

;    A signal is generated by this pal indicating that

CHIP PALO PAL22V10

; PIN DECLARATIONS
;----------------------------------------------------------------------
PIN   1    CLK20     COMBINATORIAL     ;INPUT CLOCK
PIN   2    A17       COMBINATORIAL     ;INPUT
PIN   3    A18       COMBINATORIAL     ;INPUT
PIN   4    A19       COMBINATORIAL     ;INPUT
PIN   5    A20       COMBINATORIAL     ;INPUT
PIN   6    A21       COMBINATORIAL     ;INPUT
PIN   7    A22       COMBINATORIAL     ;INPUT
PIN   8    A23       COMBINATORIAL     ;INPUT
PIN   9    A24       COMBINATORIAL     ;INPUT
PIN  10    A25       COMBINATORIAL     ;INPUT
PIN  11    /RESET    COMBINATORIAL     ;INPUT
PIN  12    GND                         ;SUPPLY RAIL
PIN  13    FC0       COMBINATORIAL     ;INPUT
PIN  14    FC1       COMBINATORIAL     ;INPUT
PIN  15    FC2       COMBINATORIAL     ;INPUT
PIN  16    /AS       COMBINATORIAL     ;OUTPUT
PIN  17    /ROMSP    COMBINATORIAL     ;OUTPUT
PIN  18    /IOSP     COMBINATORIAL     ;OUTPUT
PIN  19    /CIIN     COMBINATORIAL     ;OUTPUT
PIN  20    /RAMSP    COMBINATORIAL     ;OUTPUT
PIN  21    /ARYSP    COMBINATORIAL     ;OUTPUT
PIN  22    /AVEC     REGISTERED        ;REGISTERED OUTPUT
PIN  23    A16       COMBINATORIAL     ;INPUT
PIN  24    VCC                         ;SUPPLY RAIL
;----------------------------------------------------------------------

EQUATIONS
;  Setup FC1, FC2, AS, and A16 as inputs.
FC1.TRST = GND
FC2.TRST = GND
AS.TRST  = GND
A16.TRST = GND

;  ARRAY SPACE:  16Mb-64Mb
ARYSP  = /(FC2*FC1*FC0)*/RESET*/(/A25*/A24)

;  RAM SPACE:  1Mb-16Mb
RAMSP  = /(FC2*FC1*FC0)*/RESET*(/A25*/A24)*/(/A23*/A22*/A21*/A20)
```

```
;  I/O SPACE:  128K-1Mb
 IOSP   = /(FC2*FC1*FC0)*
 /RESET*(/A25*/A24*/A23*/A22*/A21*/A20)*/(/A19*/A18*/A17)

 ;  ROM SPACE:  OK-128K
 ROMSP  = /(FC2*FC1*FC0)*/RESET*
 /A25*/A24*/A23*/A22*/A21*/A20*/A19*/A18*/A17

 ;  CACHE INHIBIT INPUT (/CIIN):
 CIIN   = /RESET * AS * /(FC2*FC1*FC0) *
 ((/A25*/A24*/A23*/A22*/A21*/A20)*/(/A19*/A18*/A17) + /(/A25*/A24))

 ;  INTERRUPT AUTOVECTOR:
 AVEC  := /RESET * AS * (FC2 * FC1 * FC0) * (A19 * A18 * A17 * A16)

SIMULATION

TRACE_ON FC2 FC1 FC0 A25 A24 A23 A22 A21 A20 A19 A18 A17
 RESET ROMSP IOSP RAMSP ARYSP CIIN AVEC

SETF /RESET /FC2 /FC1 FC0 AS

; Check ROM space.
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check I/O space.
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18  A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19  A18 /A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19  A18  A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19 /A18 /A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19 /A18  A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19  A18 /A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19  A18  A17
CHECK /ROMSP  IOSP /RAMSP /ARYSP CIIN

; Check RAM space.
SETF /A25 /A24 /A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22  A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22 /A21  A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22  A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22  A21  A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22  A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22 /A21  A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22  A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22  A21  A20 /A19 /A18 /A17
```

```
CHECK /ROMSP /IOSP  RAMSP /ARYSP /CIIN

; Check Array space.
SETF /A25  A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP /RAMSP  ARYSP CIIN
SETF  A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP /RAMSP  ARYSP CIIN
SETF  A25  A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK /ROMSP /IOSP /RAMSP  ARYSP CIIN

; Check during RESET.
SETF RESET

; Check ROM space.
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check I/O space.
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19  A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19  A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19 /A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19  A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19  A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check RAM space.
SETF /A25 /A24 /A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22  A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22  A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check Array space.
SETF /A25  A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF  A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF  A25  A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
```

```
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check in CPU space.
SETF /RESET FC2 FC1 FC0

; Check ROM space.
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check I/O space.
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19  A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20 /A19  A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19 /A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19  A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22 /A21 /A20  A19  A18  A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check RAM space.
SETF /A25 /A24 /A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23 /A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24 /A23  A22  A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23 /A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22 /A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22  A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF /A25 /A24  A23  A22  A21  A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check Array space.
SETF /A25  A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF  A25 /A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN
SETF  A25  A24 /A23 /A22 /A21 /A20 /A19 /A18 /A17
CHECK  /ROMSP /IOSP /RAMSP /ARYSP /CIIN

; Check interrupt autovector.
SETF /RESET FC2 FC1 FC0 A19 A18 A17 A16
CLOCKF
CHECK AVEC
SETF RESET FC2 FC1 FC0 A19 A18 A17 A16
CLOCKF
CHECK /AVEC
```

```
SETF /RESET /FC2 FC1 FC0 A19 A18 A17 A16
CLOCKF
CHECK /AVEC
SETF /RESET FC2 /FC1 FC0 A19 A18 A17 A16
CLOCKF
CHECK /AVEC
SETF /RESET FC2 FC1 /FC0 A19 A18 A17 A16
CLOCKF
CHECK /AVEC
SETF /RESET FC2 FC1 FC0 /A19 A18 A17 A16
CLOCKF
CHECK /AVEC
SETF /RESET FC2 FC1 FC0 A19 /A18 A17 A16
CLOCKF
CHECK /AVEC
SETF /RESET FC2 FC1 FC0 A19 A18 /A17 A16
CLOCKF
CHECK /AVEC
SETF /RESET FC2 FC1 FC0 A19 A18 A17 /A16
CLOCKF
CHECK /AVEC

TRACE_OFF
```

C.5 PAL1A

```
TITLE UU/UM/LH/LL DECODER, DSACK GENERATOR
PATTERN PAL1
REVISION 1.3
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 8-29-90
;    PAL1 MUST BE PAL16L8-7 DC
;    -- PAL16L8-7 -- 7.5 ns
;    -- D:  ceramic package
;    -- C:  Commercial grade

;    DESCRIPTION:
;    PAL1 provides byte select generation and decoding according to the
;    following table where the mnemonics UU, UM, LH, and LL represent byte
;    selects for the next to most significant byte, next most significant
;    byte, next to least significant byte, and the least significant byte
;    of the data word:
;
;    A1 A0 SIZ1 SIZ0 | UU UM LH LL
;    ----------------+-----------
;    0  0  0    0    | 1  1  1  1
;    0  0  0    1    | 0  0  0  1
;    0  0  1    0    | 0  0  1  1
;    0  0  1    1    | 0  1  1  1
;    0  1  0    0    | 1  1  1  0
;    0  1  0    1    | 0  0  1  0
;    0  1  1    0    | 0  1  1  0
;    0  1  1    1    | 1  1  1  0
;    1  0  0    0    | 1  1  0  0
;    1  0  0    1    | 0  1  0  0
;    1  0  1    0    | 1  1  0  0
;    1  0  1    1    | 1  1  0  0
```

```
;   1  1  0   0   | 1  0  0  0
;   1  1  0   1   | 1  0  0  0
;   1  1  1   0   | 1  0  0  0
;   1  1  1   1   | 1  0  0  0
;   ----------------+-----------
;
;   Additionally, the PAL provides DSACK generation for the serial,
;   SCSI, ROM, and I/O expansion slot.

CHIP PAL1 PAL16L8
;-------------------------------------------------
PIN  1  SIZ0        COMBINATORIAL ;INPUT
PIN  2  SIZ1        COMBINATORIAL ;INPUT
PIN  3  A0          COMBINATORIAL ;INPUT
PIN  4  A1          COMBINATORIAL ;INPUT
PIN  5  /SCSIDTACK  COMBINATORIAL ;INPUT
PIN  6  /SIODTACK   COMBINATORIAL ;INPUT
PIN  7  /IO8DTACK   COMBINATORIAL ;INPUT
PIN  8  /IO16DTACK  COMBINATORIAL ;INPUT
PIN  9  /IO32DTACK  COMBINATORIAL ;INPUT
PIN 10  GND                       ;GROUND RAIL
PIN 11  /ROMDTACK   COMBINATORIAL ;INPUT
PIN 12  /UU         COMBINATORIAL ;OUTPUT
PIN 13  /UM         COMBINATORIAL ;OUTPUT
PIN 14  /LM         COMBINATORIAL ;OUTPUT
PIN 15  /LL         COMBINATORIAL ;OUTPUT
PIN 16  /DSACK0     COMBINATORIAL ;OUTPUT
PIN 17  /DSACK1     COMBINATORIAL ;OUTPUT
PIN 20  VCC                       ;SUPPLY RAIL
;-------------------------------------------------

STRING DT3 '(/SIZ1 * /SIZ0) + (A1 * A0) + (SIZ1 * A1) + (SIZ1 * SIZ0 * A0)'
STRING
DT2 '(A1 * /A0) + /A1 * (/SIZ1 * /SIZ0 + SIZ1 * SIZ0) + (/A1 * A0 * SIZ1)'
STRING DT1 '(/A1 * A0) + (/SIZ0 * /A1) + (SIZ1 * /A1)'
STRING DT0 ' /A1 * /A0 '
STRING EIGHTBITPORT 'SCSIDTACK+SIODTACK+IO8DTACK+ROMDTACK'
STRING SIXTEENBITPORT 'IO16DTACK'
STRING THIRTYTWOBITPORT 'IO32DTACK'


EQUATIONS

; Byte select equations.
UU  = (DT3)
UM  = (DT2)
LM  = (DT1)
LL  = (DT0)

; Data Strobe Acknowledge signals.
DSACK0 = EIGHTBITPORT + THIRTYTWOBITPORT
DSACK1 = SIXTEENBITPORT + THIRTYTWOBITPORT

SIMULATION

; Check byte select logic.
TRACE_ON SIZ1 SIZ0 A1 A0 UU UM LM LL
SETF /SIZ1 /SIZ0 /A1 /A0
CHECK UU UM LM LL
SETF /SIZ1 /SIZ0 /A1 A0
CHECK UU UM LM /LL
SETF /SIZ1 /SIZ0 A1 /A0
CHECK UU UM /LM /LL
SETF /SIZ1 /SIZ0 A1 A0
CHECK UU /UM /LM /LL
```

```
      SETF /SIZ1 SIZO /A1 /AO
      CHECK /UU /UM /LM LL
      SETF /SIZ1 SIZO /A1 AO
      CHECK /UU /UM LM /LL
      SETF /SIZ1 SIZO A1 /AO
      CHECK /UU UM /LM /LL
      SETF /SIZ1 SIZO A1 AO
      CHECK UU /UM /LM /LL
      SETF SIZ1 /SIZO /A1 /AO
      CHECK /UU /UM LM LL
      SETF SIZ1 /SIZO /A1 AO
      CHECK /UU UM LM /LL
      SETF SIZ1 /SIZO A1 /AO
      CHECK UU UM /LM /LL
      SETF SIZ1 /SIZO A1 AO
      CHECK UU /UM /LM /LL
      SETF SIZ1 SIZO /A1 /AO
      CHECK /UU UM LM LL
      SETF SIZ1 SIZO /A1 AO
      CHECK UU UM LM /LL
      SETF SIZ1 SIZO A1 /AO
      CHECK UU UM /LM /LL
      SETF SIZ1 SIZO A1 AO
      CHECK UU /UM /LM /LL
      TRACE_OFF

      ; Check DSACK generation logic.
      TRACE_ON SCSIDTACK SIODTACK IO8DTACK IO16DTACK IO32DTACK
       ROMDTACK DSACK1 DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      SETF  SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1  DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      SETF /SCSIDTACK  SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1  DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      SETF /SCSIDTACK /SIODTACK  IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1  DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK  ROMDTACK
      CHECK /DSACK1  DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK  IO16DTACK /IO32DTACK /ROMDTACK
      CHECK  DSACK1 /DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK  IO32DTACK /ROMDTACK
      CHECK  DSACK1  DSACKO
      SETF /SCSIDTACK /SIODTACK /IO8DTACK /IO16DTACK /IO32DTACK /ROMDTACK
      CHECK /DSACK1 /DSACKO
      TRACE_OFF
```

C.6 PAL3B

```
TITLE ADDRESS SPACE DECODER
PATTERN PAL3B
REVISION 1.3
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 1-10-92
;    PAL3B MUST BE PALCE16V8-10 pC
;    -- PALCE16V8-10 -- 10 ns
;    -- P:  plastic package
;    -- C:  Commercial grade

;    DESCRIPTION:
;    This PAL implements a state machine which controls access to the
;    EPROM array.

CHIP PAL3B PALCE16V8
; PIN DECLARATIONS
;------------------------------------------------------------------
PIN  1   CLK      COMBINATORIAL    ;INPUT CLOCK
PIN  2   /RESET   COMBINATORIAL    ;INPUT
PIN  3   /ROMSP   COMBINATORIAL    ;INPUT
PIN  4   RW       COMBINATORIAL    ;INPUT
PIN  5   /AS      COMBINATORIAL    ;INPUT
PIN  10  GND                       ;SUPPLY RAIL
PIN  14  /DBOUT   REGISTERED       ;REGISTERED OUTPUT
PIN  15  SV1      REGISTERED       ;REGISTERED OUTPUT
PIN  16  SV0      REGISTERED       ;REGISTERED OUTPUT
PIN  18  /DTACK   REGISTERED       ;REGISTERED OUTPUT
PIN  20  VCC                       ;SUPPLY RAIL
;------------------------------------------------------------------

STATE
MOORE_MACHINE
DEFAULT_BRANCH S0

;   STATE DEFINITIONS

S0 = /DBOUT *  /SV1 *  /SV0 *  /DTACK
S1 = /DBOUT *  /SV1 *   SV0 *   DTACK
S2 = /DBOUT *   SV1 *  /SV0 *  /DTACK
S3 =  DBOUT *   SV1 *   SV0 *   DTACK
S4 =  DBOUT *  /SV1 *  /SV0 +  /DTACK

;   STATE TRANSITION EQUATIONS

S0  := WAIT -> S0
    +  WRITE -> S1
    +  READ -> S3
; Write cycle
S1  := VCC -> S2

S2  := VCC -> S0

; Read cycle
S3  := VCC -> S4

S4  := VCC -> S0

CONDITIONS

WAIT    = /ROMSP + RESET
WRITE   = ROMSP * AS * /RW * /RESET
READ    = ROMSP * AS * RW * /RESET

EQUATIONS

SIMULATION
```

```
TRACE_ON RESET ROMSP RW AS DBOUT SV1 SVO DTACK

; Perform a RESET.
SETF RESET /CLK
CLOCKF
CHECK /DBOUT /SV1 /SVO /DTACK

; Hold.
SETF /RESET /ROMSP RW /AS
CLOCKF
CHECK /DBOUT /SV1 /SVO /DTACK

; Perform a read cycle.
SETF ROMSP AS RW
CLOCKF
CHECK DBOUT  SV1 SVO DTACK
CLOCKF
CHECK DBOUT /SV1 /SVO /DTACK
SETF /AS
CLOCKF
CHECK /DBOUT /SV1 /SVO /DTACK

; Perform a "write" cycle.
SETF ROMSP AS /RW
CLOCKF
CHECK /DBOUT /SV1 SVO DTACK
CLOCKF
CHECK /DBOUT SV1 /SVO /DTACK
SETF /AS
CLOCKF
CHECK /DBOUT /SV1 /SVO /DTACK
TRACE_OFF
```

C.7 PAL4

```
TITLE UU/UM/LM/LL SCRAM OE/WRITE ENCODER
PATTERN PAL4
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 5-28-90

;    PAL4 MUST BE PAL16L8-7 DC
;    -- PAL16L8-7 -- 7.5 ns
;    -- D:  ceramic package
;    -- C:  Commercial grade

;    DESCRIPTION:
;    PAL4 provides gated chip selects {UU/UM/LM/LL}CS for the SCRAM array.
;    PAL4 also provides the RAM bank select signal.

CHIP PAL4 PAL16L8

;PINS 1    2      3      4     5     6       7    8    9    10
      /UU /UM    /LM    /LL   /CS   /CSALL   A20  A21  A22  GND
;PINS 11   12     13     14    15    16       17   18   19   20
      A23 /UUCS  /UMCS  /LMCS /LLCS /BANKSEL NC   NC   NC   VCC

EQUATIONS

;  CHIP SELECT EQUATIONS
UUCS = CS*(CSALL+UU)
UMCS = CS*(CSALL+UM)
LMCS = CS*(CSALL+LM)
LLCS = CS*(CSALL+LL)
```

```
;   BANK SELECT EQUATION
BANKSEL = /A23 * /A22 * /A21 * A20

SIMULATION
; CHIP SELECT SIMULATION.
TRACE_ON UU UM LM LL CS CSALL UUCS UMCS LMCS LLCS

SETF /UU /UM /LM /LL /CS /CSALL
CHECK /UUCS /UMCS /LMCS /LLCS
SETF  UU  UM  LM  LL /CS /CSALL
CHECK /UUCS /UMCS /LMCS /LLCS
SETF  UU /UM /LM /LL  CS /CSALL
CHECK  UUCS /UMCS /LMCS /LLCS
SETF /UU  UM /LM /LL  CS /CSALL
CHECK /UUCS  UMCS /LMCS /LLCS
SETF /UU /UM  LM /LL  CS /CSALL
CHECK /UUCS /UMCS  LMCS /LLCS
SETF /UU /UM /LM  LL  CS /CSALL
CHECK /UUCS /UMCS /LMCS  LLCS
SETF  UU /UM  LM /LL  CS  CSALL
CHECK  UUCS  UMCS  LMCS  LLCS
SETF  UU /UM  LM /LL /CS  CSALL
CHECK /UUCS /UMCS /LMCS /LLCS

TRACE_OFF

; BANK SELECT SIMULATION.
TRACE_ON A23 A22 A21 A20 BANKSEL
SETF /A23 /A22 /A21 /A20
CHECK /BANKSEL
SETF /A23 /A22 /A21  A20
CHECK  BANKSEL
SETF /A23 /A22  A21 /A20
CHECK /BANKSEL
SETF /A23 /A22  A21  A20
CHECK /BANKSEL
SETF /A23  A22 /A21 /A20
CHECK /BANKSEL
SETF /A23  A22 /A21  A20
CHECK /BANKSEL
SETF /A23  A22  A21 /A20
CHECK /BANKSEL
SETF /A23  A22  A21  A20
CHECK /BANKSEL
SETF  A23 /A22 /A21 /A20
CHECK /BANKSEL
SETF  A23 /A22 /A21  A20
CHECK /BANKSEL
SETF  A23 /A22  A21 /A20
CHECK /BANKSEL
SETF  A23 /A22  A21  A20
CHECK /BANKSEL
SETF  A23  A22 /A21 /A20
CHECK /BANKSEL
SETF  A23  A22 /A21  A20
CHECK /BANKSEL
SETF  A23  A22  A21 /A20
CHECK /BANKSEL
SETF  A23  A22  A21  A20
CHECK /BANKSEL
```

TRACE_OFF

C.8 PAL5

TITLE I/O ADDRESS SPACE DECODER
PATTERN PAL5
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 5-26-90
;   PALO MUST BE PAL22V10-10 DC
;   -- PAL22V10-10 -- 10 ns
;   -- D: ceramic package
;   -- C: Commercial grade

;   DESCRIPTION:
;   This PAL decodes I/O space selects for I/O devices.
;   The memory map is described elsewhere.
;
;   The chip also determines the I/O/CPU bus data transceiver direction.

CHIP PAL5 PAL22V10

;PINS 1   2   3   4   5   6   7   8   9   10   11   12 13
      A8   A9 A10 A11 A12   A13   A14   A15 A16   A17 A18   GND A19
;PINS 14 15   16   17   18   19   20 21   22   23   24   (25)
      RW /OER /SCSI1BUF   /SCSI1 /SCSIDBA /SIO1 NC /IOSP /AS   /OET VCC NC

EQUATIONS

; Additional inputs.
IOSP.TRST = GND
AS.TRST   = GND
RW.TRST   = GND

; First SCSI device (SCSI1).
SCSI1 = /A19 * /A18 * A17 * /A16 * /A15 * /A14 * /A13 * /A12 *
/A11 * /A10 * /A9 * /A8 * AS * IOSP

; SCSI device buffer memory (SCSI1BUF).
SCSI1BUF = /A19 * /A18 * A17 * /A16 * (A15 * /A14 * /A13 * /A12 +
   /A15 * (A12 + A13 + A14))

; SCSI DBA mode control register (SCSIDBA).
SCSIDBA = /A19 * /A18 * A17 * /A16 * /A15 * /A14 * /A13 * /A12 *
/A11 * /A10 * /A9 * A8 * AS * IOSP

; First serial I/O device (SIO1).
SIO1 = /A19 * /A18 * A17 * /A16 * /A15 * /A14 * /A13 * /A12 *
 A11 * /A10 * /A9 * /A8 * AS * IOSP

; Data from CPU to I/O bus direction.
OER   = /RW * AS * IOSP

; Data from I/O bus to CPU bus direction.
OET   = RW * AS * IOSP

SIMULATION

TRACE_ON A19 A18 A17 A16 A15 A14 A13 A12 A11 A10 A9 A8 AS IOSP
 SCSI1 SCSIDBA SIO1

SETF IOSP
SETF /A19 /A18 A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10 /A9 /A8 AS
CHECK SCSI1 /SCSI1BUF /SCSIDBA /SIO1
SETF /A19 /A18 A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10 /A9 A8
CHECK /SCSI1 /SCSI1BUF SCSIDBA /SIO1

```
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10  A9 /A8
CHECK /SCSI1 /SCSI1BUF /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10  A9  A8
CHECK /SCSI1 /SCSI1BUF /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12  A11 /A10 /A9 /A8
CHECK /SCSI1 /SCSI1BUF /SCSIDBA  SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13  A12 /A11 /A10 /A9 /A8
CHECK /SCSI1  SCSI1BUF /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13  A12 /A11 /A10 /A9 /A8
CHECK /SCSI1 /SCSIDBA /SIO1

SETF /IOSP
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10 /A9 /A8 AS
CHECK /SCSI1 /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10 /A9  A8
CHECK /SCSI1 /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10  A9 /A8
CHECK /SCSI1 /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12 /A11 /A10  A9  A8
CHECK /SCSI1 /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13 /A12  A11 /A10 /A9 /A8
CHECK /SCSI1 /SCSIDBA /SIO1
SETF /A19 /A18  A17 /A16 /A15 /A14 /A13  A12 /A11 /A10 /A9 /A8
CHECK /SCSI1 /SCSIDBA /SIO1
TRACE_OFF

TRACE_ON RW AS IOSP OER OET
SETF IOSP
SETF RW /AS
CHECK /OER /OET
SETF RW  AS
CHECK /OER OET
SETF /RW /AS
CHECK /OER /OET
SETF /RW AS
CHECK OER /OET

SETF /IOSP
SETF RW /AS
CHECK /OER /OET
SETF RW  AS
CHECK /OER /OET
SETF /RW /AS
CHECK /OER /OET
SETF /RW AS
CHECK /OER /OET
TRACE_OFF
```

C.9 PAL7

```
TITLE ARRAY BUS INTERFACE
PATTERN PAL7
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 3-19-91
CHIP PAL7 PAL22V10
```

```
;    PAL7 MUST BE PAL22V10-10 DC
;    -- PAL22V10-10 -- 10 ns
;    -- D:  ceramic package
;    -- C:  Commercial grade

; PIN/NODE DECLARATIONS
; -------- ------------
;
; P/N  #  NAME          PAIRED WITH PIN      STORAGE         COMMENTS
;----  --- ------------  -------------------- -------------   --------
PIN   12 GND                                                 ; SUPPLY RAIL
PIN   24 VCC                                                 ; SUPPLY RAIL

PIN    1  /STERM0                            COMBINATORIAL   ; INPUT
PIN    2  /STERM1                            COMBINATORIAL   ; INPUT
PIN    3  /STERM2                            COMBINATORIAL   ; INPUT
PIN    4  /STERM3                            COMBINATORIAL   ; INPUT
PIN    5  /STERM4                            COMBINATORIAL   ; INPUT
PIN    6  /STERM5                            COMBINATORIAL   ; INPUT
PIN    7  /STERM6                            COMBINATORIAL   ; INPUT
PIN    8  /STERM7                            COMBINATORIAL   ; INPUT

PIN   13 /SLOTEN0                            COMBINATORIAL   ; INPUT
PIN   17 /SLOTEN1                            COMBINATORIAL   ; INPUT
PIN   18 /SLOTEN2                            COMBINATORIAL   ; INPUT
PIN   19 /SLOTEN3                            COMBINATORIAL   ; INPUT
PIN   20 /SLOTEN4                            COMBINATORIAL   ; INPUT
PIN   21 /SLOTEN5                            COMBINATORIAL   ; INPUT
PIN   22 /SLOTEN6                            COMBINATORIAL   ; INPUT
PIN   23 /SLOTEN7                            COMBINATORIAL   ; INPUT

PIN   16 /ARYST                              COMBINATORIAL   ; OUTPUT

PIN    9  /ARYSP                             COMBINATORIAL   ; INPUT
PIN   10 /AS                                 COMBINATORIAL   ; INPUT
PIN   11 RW                                  COMBINATORIAL   ; INPUT
PIN   14 /OER                                COMBINATORIAL   ; OUTPUT
PIN   15 /OET                                COMBINATORIAL   ; OUTPUT
;---- --- ------------  -------------------- -------------   --------

EQUATIONS
; DATA BUFFER CONTROL

OER = ARYSP * AS * /RW
OET = ARYSP * AS *  RW

; STERM CONTROL
ARYST = (SLOTEN0*STERM0)+(SLOTEN1*STERM1)+(SLOTEN2*STERM2)+(SLOTEN3*STERM3)
      + (SLOTEN4*STERM4)+(SLOTEN5*STERM5)+(SLOTEN6*STERM6)+(SLOTEN7*STERM7)

SLOTEN1.TRST = GND
SLOTEN2.TRST = GND
SLOTEN3.TRST = GND
SLOTEN4.TRST = GND
SLOTEN5.TRST = GND
SLOTEN6.TRST = GND
SLOTEN7.TRST = GND

SIMULATION
TRACE_ON ARYSP AS RW OER OET
SETF /ARYSP /AS /RW
CHECK /OER /OET
SETF /ARYSP /AS  RW
CHECK /OER /OET
SETF /ARYSP  AS /RW
CHECK /OER /OET
SETF /ARYSP  AS  RW
```

```
CHECK /OER /OET
SETF  ARYSP /AS /RW
CHECK /OER /OET
SETF  ARYSP /AS  RW
CHECK /OER /OET
SETF  ARYSP  AS /RW
CHECK  OER /OET
SETF  ARYSP  AS  RW
CHECK /OER  OET
TRACE_OFF

TRACE_ON ARYST SLOTEN0 SLOTEN1 SLOTEN2 SLOTEN3 SLOTEN4 SLOTEN5 SLOTEN6
 SLOTEN7 STERM0 STERM1 STERM2 STERM3 STERM4 STERM5 STERM6 STERM7
SETF /SLOTEN0 /SLOTEN1 /SLOTEN2 /SLOTEN3 /SLOTEN4 /SLOTEN5 /SLOTEN6 /SLOTEN7
SETF /STERM0 /STERM1 /STERM2 /STERM3 /STERM4 /STERM5 /STERM6 /STERM7
CHECK /ARYST
SETF SLOTEN0 STERM0
CHECK ARYST
SETF /STERM0 SLOTEN1 STERM1
CHECK ARYST
SETF /STERM1 SLOTEN2 STERM2
CHECK ARYST
SETF /STERM2 SLOTEN3 STERM3
CHECK ARYST
SETF /STERM3 SLOTEN4 STERM4
CHECK ARYST
SETF /STERM4 SLOTEN5 STERM5
CHECK ARYST
SETF /STERM5 SLOTEN6 STERM6
CHECK ARYST
SETF /STERM6 SLOTEN7 STERM1
CHECK ARYST
TRACE_OFF
```

C.10 PAL12

```
TITLE ESCC (SIO) CONTROLLER STATE MACHINE
PATTERN PAL12
REVISION 1.0
AUTHOR JON MELLOTT
COMPANY ARRAY PROJECT
DATE 9-13-90
;    PAL12 MUST BE PAL22V10-0 DC
;    -- PAL22V10-10 DC -- 10 ns
;    -- D:  ceramic package
;    -- C:  Commercial grade

;    DESCRIPTION:
;    This PAL implements the controller state machine for the serial
;    I/O module.
;
;    Additionally, this PAL controls access to a register which
;    is used to control the SCSI state machine.  Access to the register
;    is requested using SCSIDBA.  The access is terminated using
;    the regular SIODTACK.

CHIP PAL12 PAL22V10

; PIN DECLARATIONS
;-----------------------------------------------------
PIN 1  CLK20    COMBINATORIAL  ;INPUT
```

```
PIN   2   /SIO1    COMBINATORIAL   ;INPUT
PIN   3   RW       COMBINATORIAL   ;INPUT
PIN   4   /SCSIDBA COMBINATORIAL   ;INPUT
PIN  12   GND                      ;SUPPLY RAIL
PIN  14   /RD      REGISTERED       ;OUTPUT
PIN  15   /WR      REGISTERED       ;OUTPUT
PIN  16   /CE      REGISTERED       ;OUTPUT
PIN  17   /OER     REGISTERED       ;OUTPUT
PIN  18   /OET     REGISTERED       ;OUTPUT
PIN  19   /DTACK   REGISTERED       ;OUTPUT
PIN  20   /SV0     REGISTERED       ;OUTPUT
PIN  21   /SV1     REGISTERED       ;OUTPUT
PIN  22   /EN      REGISTERED       ;OUTPUT
PIN  24   VCC                      ;SUPPLY RAIL
;--------------------------------------------------
STATE
MOORE_MACHINE
DEFAULT_BRANCH S0
START_UP := POWER_UP -> S0
;   STATE DEFINITIONS
;   -----------------
; Power-up & standby.
S0 = /RD * /WR * /CE * /OER * /OET * /DTACK * /EN * /SV0 * /SV1

; Read cycle.
R0 = /RD * /WR * /CE * /OER * /OET * /DTACK * /EN *  SV0 * /SV1
R1 = /RD * /WR *  CE * /OER * /OET * /DTACK * /EN * /SV0 * /SV1
R2 =  RD * /WR *  CE *  OER * /OET * /DTACK * /EN * /SV0 * /SV1
R3 =  RD * /WR *  CE *  OER * /OET * /DTACK * /EN *  SV0 * /SV1
R4 =  RD * /WR *  CE *  OER * /OET * /DTACK * /EN *  SV0 *  SV1
R5 =  RD * /WR *  CE *  OER * /OET *  DTACK * /EN * /SV0 *  SV1

; Write cycle.
W0 = /RD * /WR * /CE * /OER * /OET * /DTACK * /EN * /SV0 *  SV1
W1 = /RD * /WR *  CE * /OER *  OET * /DTACK * /EN * /SV0 *  SV1
W2 = /RD *  WR *  CE * /OER *  OET * /DTACK * /EN *  SV0 *  SV1
W3 = /RD *  WR *  CE * /OER *  OET * /DTACK * /EN *  SV0 * /SV1
W4 = /RD *  WR *  CE * /OER *  OET *  DTACK * /EN * /SV0 * /SV1

; System I/O register access cycle.
B0 = /RD * /WR * /CE * /OER * /OET * /DTACK *  EN *  SV0 *  SV1
B1 = /RD * /WR * /CE * /OER * /OET *  DTACK * /EN * /SV0 * /SV1

;   STATE TRANSITION EQUATIONS
;   WAIT FOR CYCLE TO START STATE
S0  := WAIT      -> S0
     + GO_READ   -> R0
     + GO_WRITE  -> W0
     + GO_BUFFER -> B0
;   READ CYCLE
R0  := VCC -> R1
R1  := VCC -> R2
R2  := VCC -> R3
R3  := VCC -> R4
R4  := VCC -> R5
R5  := VCC -> S0
;   WRITE CYCLE
W0  := VCC -> W1
W1  := VCC -> W2
W2  := VCC -> W3
W3  := VCC -> W4
```

```
W4 := VCC -> S0
; REGISTER ACCESS CYCLE
B0 := VCC -> B1
B1 := VCC -> S0
CONDITIONS
;  HOLD AT S0
WAIT    = /SIO1 * /SCSIDBA
; START CYCLE
GO_READ = SIO1 * RW
GO_WRITE = SIO1 * /RW
; START REGISTER ACCESS CYCLE
GO_BUFFER = /SIO1 * SCSIDBA

EQUATIONS

SIMULATION

TRACE_ON CLK20 SIO1 SCSIDBA RW RD WR CE OER OET DTACK SVO SV1

SETF /SIO1 /SCSIDBA RW /CLK20
CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK /SVO /SV1
; HOLD
CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK /SVO /SV1
; READ
SETF SIO1 RW
CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK  SVO /SV1
CLOCKF
CHECK /RD /WR  CE /OER /OET /DTACK /SVO /SV1
CLOCKF
CHECK  RD /WR  CE  OER /OET /DTACK /SVO /SV1
CLOCKF
CHECK  RD /WR  CE  OER /OET /DTACK  SVO /SV1
CLOCKF
CHECK  RD /WR  CE  OER /OET /DTACK  SVO  SV1
CLOCKF
CHECK  RD /WR  CE  OER /OET  DTACK /SVO  SV1
CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK /SVO /SV1

SETF /SIO1 RW

CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK /SVO /SV1
; WRITE
SETF SIO1 /RW
CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK /SVO  SV1
CLOCKF
CHECK /RD /WR  CE /OER  OET /DTACK /SVO  SV1
CLOCKF
CHECK /RD  WR  CE /OER  OET /DTACK  SVO  SV1
CLOCKF
CHECK /RD  WR  CE /OER  OET /DTACK  SVO /SV1
CLOCKF
```

```
CHECK /RD  WR  CE /OER  OET  DTACK /SVO /SV1
CLOCKF
CHECK /RD /WR /CE /OER /OET /DTACK /SVO /SV1
TRACE_OFF
```

Appendix D

INVESTIGATOR SOURCE CODE

D.1 Link Specification File: BACKPLAN.LNK

```
* Original interrupt vector table and interrupt service
* routines. Modified PLC startup source code.
LINK    isr                    * Exception vector table, ISRs.
LINK    postinit               * POST code.
LINK    myrtl\startup\startup  * My startup code merged with PLC's.

* Standard PLC runtime libraries.
LINK    myrtl\rtl2nn           * Main run-time library.
LINK    myrtl\fplib2nn()       * Floating point library.

* The following comprise the routines which make up the
* PLC {SYSTEM|SYS2YN|SYS2NN}.LTX run-time libraries.
LINK    myrtl\sstkck\sstkck()     * Stack checker.
LINK    myrtl\ssiginit\ssiginit() * Signal handler.
LINK    myrtl\sdefhndl\sdefhndl()
LINK    myrtl\sread\sread()
LINK    myrtl\swrite\swrite()
LINK    myrtl\ssbrk\ssbrk()
LINK    myrtl\sfsinit\sfsinit()   * File system init.

* The following are my own modules.
LINK    backplan        * Main program. Also replaces hardware.
LINK    monitor         * Monitor program.
LINK    escc            * ESCC routines.
LINK    sbic            * SBIC routines.
LINK    convert         * QRNS conversion engine, init.

ORG     $400
SECTION 0,1
ORG     $100000
SECTION 2,13,14,15,3,4,5
STACKSZ EQU      $1FFFFF-*
.stack  DS.B     STACKSZ
END
```

D.2 Basic Type Definitions: BASETYPE.H

```
/*
  InvestiGATOR Backplane Firmware
  ------------- --------- --------

  Major Rev: 0
  Minor Rev: 0
  Date: 2/6/92
```

```
Author: Jon Mellott

High Speed Digital Architecture Laboratory
University of Florida
405 CSE Building
Gainesville, FL  32611

This header file contains basic type definitions.
*/

/* General types. */

typedef unsigned char      BOOL;
typedef unsigned char      BYTE;
typedef unsigned short int WORD;
typedef long               LONG;
typedef unsigned long      DWORD;
```

D.3   I/O Constants: INVESTIO.INC

```
; -------
; EQUATES
; -------

; ***ASCII CONSTANTS***
XON      EQU     $11        ; ^Q
XOFF     EQU     $13        ; ^S
XONBR    EQU     $88        ; Bit reversed XON.
XOFFBR   EQU     $C8        ; Bit reversed XOFF.

; ***ESCC CONSTANTS***
; Note: due to a major FUBAR, the bus bits on the ESCC are *reversed*.
; Thus, all arguments written to the ESCC must be bit reversed.
; All constants are pre-reversed. Data is reversed on the fly...

; Serial port register locations.
SIOA_DATA EQU     $20803
SIOA_CTRL EQU     $20802
SIOB_DATA EQU     $20801
SIOB_CTRL EQU     $20800

; Serial port register offsets.
R0       EQU     $0
R1       EQU     $80
R2       EQU     $40
R3       EQU     $C0
R4       EQU     $20
R5       EQU     $A0
R6       EQU     $60
R7       EQU     $E0
R8       EQU     $10
R9       EQU     $90
R10      EQU     $50
R11      EQU     $D0
R12      EQU     $30
R13      EQU     $B0
R14      EQU     $70
R15      EQU     $F0

; Write register 0 commands:
WRORTIP  EQU     $14        ; Reset Tx interrupt pending ($28).
```

```
; Read register 3 interrupt pending bit positions.
RR3ARIP   EQU        2            ; Channel A Rx IP Mask.
RR3ATIP   EQU        3            ; Channel A Tx IP Mask.
RR3AISIP  EQU        4            ; Channel A Ext/Sta IP Mask.
RR3BRIP   EQU        5            ; Channel A Rx IP Mask.
RR3BTIP   EQU        6            ; Channel A Tx IP Mask.
RR3BISIP  EQU        7            ; Channel A Ext/Sta IP Mask.

; ESCC character buffer lengths.
ESCCBUFL  EQU        128          ; ESCC buffer length.

; ESCC character buffer index mask (tied to ESCCBUFL).
ESCCBUFW  EQU        127          ; ESCC buffer index mask.

; ESCC character buffer fill threshold for XOFF.
ESCCXOFT  EQU        96           ; Threshold for XOFF.

; ESCC XON/XOFF request flag definitions.
NOXREQ    EQU        0            ; No XON/XOFF request.
XOFFREQ   EQU        1            ; XOFF request flag.
XONREQ    EQU        2            ; XON request flag.
RCVXOFF   EQU        3            ; Receiver is XOFF, no XON/XOFF request.

; ***SBIC CONSTANTS***
; DBA mode enable register location.
SCSI_DBA  EQU        $20100

; Base location of SCSI RAM buffer.
SCSI_RAM  EQU        $21000

; SCSI port base address.
SCSI_PORT EQU        $20000

; SCSI port register displacements.
SCS_OWNID EQU        0
SCS_CTRLR EQU        1
SCS_TIMOT EQU        2
SCS_TSECT EQU        3
SCS_THEAD EQU        4
SCS_TCYL1 EQU        5
SCS_TCYL0 EQU        6
SCS_LADR3 EQU        7
SCS_LADR2 EQU        8
SCS_LADR1 EQU        9
SCS_LADR0 EQU        $A
SCS_SECTN EQU        $B
SCS_HEADN EQU        $C
SCS_CYLN1 EQU        $D
SCS_CYLN0 EQU        $E
SCS_TLUNN EQU        $F
SCS_CMDPR EQU        $10
SCS_SYNTR EQU        $11
SCS_TRCR2 EQU        $12
SCS_TRCR1 EQU        $13
SCS_TRCR0 EQU        $14
SCS_DSTID EQU        $15
SCS_SRCID EQU        $16
SCS_STATS EQU        $17
SCS_CMDRG EQU        $18
SCS_DATAR EQU        $19
SCS_AUXSR EQU        $1F
```

D.4 Base Firmware: BACKPLAN.C

```
#include<stdio.h>
#include<stdlib.h>

/*
  InvestiGATOR Backplane Firmware
  ------------ --------- --------

  Major Rev: 0
  Minor Rev: 0
  Date: 2/6/92

  Author: Jon Mellott

  High Speed Digital Architecture Laboratory
  University of Florida
  405 CSE Building
  Gainesville, FL  32611

  This software constitutes the basic operating system of the
  InvestiGATOR backplane.

  The InvestiGATOR backplane is configured as follows:

  MC68030 processor operating at 20 MHz.
  128KB ROM, base address $0, eight bit organization.
  1MB SCRAM, burst mode read access supported,
       base address $100000, thirty-two bit organization.
  Am33C93A SCSI Bus Interface Controller, single-ended, eight-bit,
       SCSI-2, synchronous up to 5.0 MB/s.
  Z85C30 Enhanced Serial Communications Controller, two RS-232
       serial ports, up to 19.2 kbps.
*/

/* ------- ------- */
/* INCLUDE SECTION */
/* ------- ------- */

/* #include<stdio.h>  */
/* #include<stdlib.h> */

/* ----- ----------- */
/* MACRO DEFINITIONS */
/* ----- ----------- */
#define PUT8(addr,val)  (*(unsigned char *)(addr))=val
#define GET8(addr)      (*(unsigned char *)(addr))

/* --------- ------- */
/* CONSTANTS SECTION */
/* --------- ------- */

/* Logicals */
#define TRUE     1
#define FALSE    0

/* ASCII Constants */
#define XON      0x11 /* ^Q */
#define XOFF     0x13 /* ^S */

/* ---- ----------- */
/* TYPE DEFINITIONS */
/* ---- ----------- */

#include "basetype.h"

/* -------- ---------- */
/* EXTERNAL PROCEDURES */
```

```
/* -------- ---------- */
extern void SerialPortInit(); /* ESCC.C */
extern void SCSIPortInit();   /* SBIC.C */

/* ------- ------------ */
/* STATIC DECLARATIONS */
/* ------- ------------ */

/* --------- -------- */
/* PROCEDURE DIVISION */
/* --------- -------- */


/* ---- --------- */
/* MAIN PROCEDURE */
/* ---- --------- */

main(void)
{
  /* Print banner. */
  printf("\n\nInvestiGATOR Firmware C Monitor version 0.0 5/29/92F");
  printf("\nHigh Speed Digital Architecture Laboratory");
  printf("\nUniversity of Florida");

  /* Initialize the serial port. */
  SerialPortInit;

  /* Initialize the SCSI port. */
  SCSIPortInit;

  /* Main program loop. */
  while(TRUE) ;
}
```

D.5 QRNS Conversion Code: CONVERT.C

```
#define TESTCONVERT 0 /* Enable standalone compilation for testing. */
#include <stdio.h>
#include <stdlib.h>

/* Program: Conversion Engine
   Major Rev: 0
   Minor Rev: 0

   Author: Jon Mellott
   Date:   2/14/92F

   Description:
   This module contains the necessary components for forward
   conversion from Gaussian integers to the QRNS and
   conversion from QRNS to Gaussian integers.
*/

/* ---------- */
/* CONSTANTS */
/* ---------- */

/* Logicals */
#define TRUE 1
#define FALSE 0

/* Conversion system parameters. */
```

```
#define iPrime1 113
#define iPrime2 109
#define iPrime3 101
#define lM1       (long)11009
#define lM2       (long)11413
#define lM3       (long)12317
#define iM1Inv    73
#define iM2Inv    17
#define iM3Inv    20
#define lM        (long)1244017
#define iJHat1    15
#define iJHat2    33
#define iJHat3    10
#define iJHatI1 98
#define iJHatI2 76
#define iJHatI3 91
#define lTwoI1  (long)57
#define lTwoI2  (long)55
#define lTwoI3  (long)51


/* ---- ----------- */
/* TYPE DEFINITIONS */
/* ---- ----------- */

/* Define basic types. */
#include "basetype.h"

/* Define RNS three tuple. */
typedef struct _RNS {
    int iR1,iR2,iR3;
} RNS;
typedef RNS *           RNSPTR;


/* Define Gaussian integer. */
typedef struct _GAUSSIAN {
    long lR;
    long lI;
} GAUSSIAN;
typedef GAUSSIAN *      GAUSSPTR;


/* ------- ---------- */
/* STATIC VARIABLES */
/* ------- ---------- */

/* Mod p tables */
static BYTE    byMod1[0x200];
static BYTE    byMod2[0x200];
static BYTE    byMod3[0x200];

/* Forward conversion tables. */
static BYTE    byFC1a[0x80];
static BYTE    byFC1b[0x80];
static BYTE    byFC1c[0x80];
static BYTE    byFC2a[0x80];
static BYTE    byFC2b[0x80];
static BYTE    byFC2c[0x80];
static BYTE    byFC3a[0x80];
static BYTE    byFC3b[0x80];
static BYTE    byFC3c[0x80];

/* QRNS to CRNS tables. */
```

```
static BYTE    byQtoC1[0x180];
static BYTE    byQtoC2[0x180];
static BYTE    byQtoC3[0x180];
static BYTE    byQtoC1i[0x180];
static BYTE    byQtoC2i[0x180];
static BYTE    byQtoC3i[0x180];

/* CRT tables. */
static long    lCRT1[0x80];
static long    lCRT2[0x80];
static long    lCRT3[0x80];

/* Multiplication by j_hat and j_hat^(-1) tables. */
static BYTE    byMJH1[0x80];
static BYTE    byMJH2[0x80];
static BYTE    byMJH3[0x80];
static BYTE    byMJHI1[0x80];
static BYTE    byMJHI2[0x80];
static BYTE    byMJHI3[0x80];

/* ---------- -------- */
/* PROCEDURE DIVISION */
/* ---------- -------- */
/* -------------------------------------
   Procedure: ConEngInit

   Parameters: None

   Description:
   This procedure creates the tables necessary for conversion between
   Gaussian integers and QRNS.

   Returns: Nothing
*/
void ConEngInit()
{
  int iTemp;
  long lTemp;

  /* Generate mod p tables. */
  for (iTemp=0; iTemp<0x200; ++iTemp) {
    byMod1[iTemp]=(BYTE)(iTemp % iPrime1);
  }
  for (iTemp=0; iTemp<0x200; ++iTemp) {
    byMod2[iTemp]=(BYTE)(iTemp % iPrime2);
  }
  for (iTemp=0; iTemp<0x200; ++iTemp) {
    byMod3[iTemp]=(BYTE)(iTemp % iPrime3);
  }
  /* Generate forward conversion tables. */
  for (lTemp=0; lTemp<0x80; ++lTemp) {
    byFC1a[lTemp]=(BYTE)(lTemp % iPrime1);
    byFC1b[lTemp]=(BYTE)(lTemp*0x80 % iPrime1);
    byFC1c[lTemp]=(BYTE)(lTemp*0x4000 % iPrime1);
    byFC2a[lTemp]=(BYTE)(lTemp % iPrime2);
    byFC2b[lTemp]=(BYTE)(lTemp*0x80 % iPrime2);
    byFC2c[lTemp]=(BYTE)(lTemp*0x4000 % iPrime2);
    byFC3a[lTemp]=(BYTE)(lTemp % iPrime3);
```

```
      byFC3b[lTemp]=(BYTE)(lTemp*0x80 % iPrime3);
      byFC3c[lTemp]=(BYTE)(lTemp*0x4000 % iPrime3);
  }
  /* Generate QRNS to CRNS tables. */
  for (lTemp=0; lTemp<0x180; ++lTemp) {
     byQtoC1[lTemp]=(BYTE)(lTwoI1*lTemp % iPrime1);
     byQtoC2[lTemp]=(BYTE)(lTwoI2*lTemp % iPrime2);
     byQtoC3[lTemp]=(BYTE)(lTwoI3*lTemp % iPrime3);
     byQtoC1i[lTemp]=(BYTE)((long)iJHatI1*lTwoI1*lTemp % iPrime1);
     byQtoC2i[lTemp]=(BYTE)((long)iJHatI2*lTwoI2*lTemp % iPrime2);
     byQtoC3i[lTemp]=(BYTE)((long)iJHatI3*lTwoI3*lTemp % iPrime3);
  }
  /* Generate CRT tables. */
  for (lTemp=0; lTemp<0x80; ++lTemp) {
     lCRT1[lTemp]=(lM1*(iM1Inv*lTemp % iPrime1)) % lM;
     lCRT2[lTemp]=(lM2*(iM2Inv*lTemp % iPrime2)) % lM;
     lCRT3[lTemp]=(lM3*(iM3Inv*lTemp % iPrime3)) % lM;
  }
  /* Generate multiplication by j_hat and j_hat^{-1} mod p tables. */
  for (lTemp=0; lTemp<0x30; ++lTemp) {
     byMJH1[lTemp]=(BYTE)(iJHat1*lTemp % iPrime1);
     byMJH2[lTemp]=(BYTE)(iJHat2*lTemp % iPrime2);
     byMJH3[lTemp]=(BYTE)(iJHat3*lTemp % iPrime3);
     byMJHI1[lTemp]=(BYTE)(iJHatI1*lTemp % iPrime1);
     byMJHI2[lTemp]=(BYTE)(iJHatI2*lTemp % iPrime2);
     byMJHI3[lTemp]=(BYTE)(iJHatI3*lTemp % iPrime3);
  }
  /* Done. */
  return ;
}

/* ------------------------------------
   Procedure: ForwardConvert

   Parameters:
   -----------
   gN  -- Gaussian integer.
   RZ  -- Z QRNS channel.
   RZS -- Z* QRNS channel.
   -----------
   Description:
   This procedure takes a Gaussian integer as input and produces a
   QRNS representation.

   Returns: Nothing
*/
void ForwardConvert(GAUSSPTR gN, RNSPTR RZ, RNSPTR RZS)
{
  RNS RReal,RImag;
  long lTemp;
  long lW1,lW2,lW3;

  /* Convert Gaussian integer to CRNS. */
  lW1=gN->lR & (long)0x7F;
  lW2=(gN->lR & (long)0x3F80) >> 7;
  lW3=(gN->lR & (long)0x1FC000) >> 14;

  lTemp=(long)byFC1a[lW1] + (long)byFC1b[lW2] + (long)byFC1c[lW3];
```

```
    RReal.iR1 = (int)byMod1[lTemp];

    lTemp=(long)byFC2a[lW1] + (long)byFC2b[lW2] + (long)byFC2c[lW3];
    RReal.iR2 = (int)byMod2[lTemp];

    lTemp=(long)byFC3a[lW1] + (long)byFC3b[lW2] + (long)byFC3c[lW3];
    RReal.iR3 = (int)byMod3[lTemp];

    lW1=gN->lI & (long)0x7F;
    lW2=(gN->lI & (long)0x3F80) >> 7;
    lW3=(gN->lI & (long)0x1FC000) >> 14;

    lTemp=(long)byFC1a[lW1] + (long)byFC1b[lW2] + (long)byFC1c[lW3];
    RImag.iR1 = (int)byMod1[lTemp];

    lTemp=(long)byFC2a[lW1] + (long)byFC2b[lW2] + (long)byFC2c[lW3];
    RImag.iR2 = (int)byMod2[lTemp];

    lTemp=(long)byFC3a[lW1] + (long)byFC3b[lW2] + (long)byFC3c[lW3];
    RImag.iR3 = (int)byMod3[lTemp];

    /* Convert CRNS to QRNS. */
    RZ->iR1=(int)byMod1[RReal.iR1+(int)byMJH1[RImag.iR1]];
    RZ->iR2=(int)byMod2[RReal.iR2+(int)byMJH2[RImag.iR2]];
    RZ->iR3=(int)byMod3[RReal.iR3+(int)byMJH3[RImag.iR3]];
    RZS->iR1=(int)byMod1[RReal.iR1+(int)byMJHI1[RImag.iR1]];
    RZS->iR2=(int)byMod2[RReal.iR2+(int)byMJHI2[RImag.iR2]];
    RZS->iR3=(int)byMod3[RReal.iR3+(int)byMJHI3[RImag.iR3]];

    /* Done. */
    return ;
}

/* ------------------------------------
    Procedure: QRNSCRT

    Parameters:
    -----------
    RZ  -- Z QRNS channel.
    RZS -- Z* QRNS channel.
    gN  -- Gaussian integer.
    -----------
    Description:
    This procedure takes a QRNS input and produces the Gaussian integer
    representation of that output.

    Returns: Nothing
*/
void QRNSCRT(RNSPTR RZ, RNSPTR RZS, GAUSSPTR gN)
{
  RNS RReal,RImag;
  int iTemp;
  long lTemp;

  /* Produce CRNS from QRNS. */
  iTemp=RZ->iR1+RZS->iR1;
  RReal.iR1=(int)byQtoC1[iTemp];
  iTemp=RZ->iR1+iPrime1-RZS->iR1;
  RImag.iR1=(int)byQtoC1i[iTemp];

  iTemp=RZ->iR2+RZS->iR2;
  RReal.iR2=(int)byQtoC2[iTemp];
  iTemp=RZ->iR2+iPrime2-RZS->iR2;
  RImag.iR2=(int)byQtoC2i[iTemp];
```

```
    iTemp=RZ->iR3+RZS->iR3;
    RReal.iR3=(int)byQtoC3[iTemp];
    iTemp=RZ->iR3+iPrime3-RZS->iR3;
    RImag.iR3=(int)byQtoC3i[iTemp];

    /* Perform CRT. */
    /* Compute partial sum. */
    gN->lR=lCRT1[RReal.iR1]+lCRT2[RReal.iR2]+lCRT3[RReal.iR3];
    gN->lI=lCRT1[RImag.iR1]+lCRT2[RImag.iR2]+lCRT3[RImag.iR3];
    /* Perform modular reduction. */
    if ((lTemp=gN->lR-lM) >= 0) {
      gN->lR=lTemp;
      if ((lTemp=gN->lR-lM) >= 0) {
        gN->lR=lTemp;
      }
    }
    if ((lTemp=gN->lI-lM) >= 0) {
      gN->lI=lTemp;
      if ((lTemp=gN->lI-lM) >= 0) {
        gN->lI=lTemp;
      }
    }
    /* Done. */
    return ;
}

/* Test generation code. */
#if TESTCONVERT

main()
{
  RNS RZ,RZS;
  GAUSSIAN gN,gM;
  long lTemp,lTemp2;

  printf("\nTest Fixture for Forward Conversion/CRT Engine Code.");
  printf("\nv0.0, 2/15/92Sat");
  printf("\nJ. Mellott");

  printf("\n\nInitializing tables...");
  ConEngInit();

  printf("\n\nTesting conversion engine (real conversion only)...");
  for (lTemp=0; lTemp<lM; ++lTemp) {
    gN.lR=lTemp;
    gN.lI=(long)0;
    ForwardConvert(&gN,&RZ,&RZS);
    gN.lR=0;
    gN.lI=0;
    QRNSCRT(&RZ,&RZS,&gN);
    if (gN.lR != lTemp || gN.lI!=0) {
      printf("\nFault at %li.\n",lTemp);
    }
    if ((lTemp % 10000) == 0) printf(".");
  }
  printf("\n\nTesting conversion engine (imaginary conversion only)...");
  for (lTemp=0; lTemp<lM; ++lTemp) {
    gN.lR=(long)0;
    gN.lI=lTemp;
```

```
                :

   ForwardConvert(&gN,&RZ,&RZS);

   QRNSCRT(&RZ,&RZS,&gN);
   if (gN.1R!=0 || gN.1I != 1Temp) {
     printf("\nFault at %li.\n",1Temp);
   }
   if ((1Temp % 10000) == 0) printf(".");
  }
  printf("\n\nTesting conversion engine (incomplete complex testing)...");
  for (1Temp=0; 1Temp<1M; 1Temp+=1111) {
    for (1Temp2=0; 1Temp2<1M; 1Temp2+=1234) {
      gN.1R=1Temp;
      gN.1I=1Temp2;
      ForwardConvert(&gN,&RZ,&RZS);

      QRNSCRT(&RZ,&RZS,&gN);
      if (gN.1R != 1Temp || gN.1I != 1Temp2) {
        printf("\nFault at %li.\n",1Temp);
      }
    }
    printf(".");
  }

}
#endif
```

D.6 Monitor: MONITOR.C

```c
#define TESTVERSION   0 /* Debugging control for conditional compilation. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

/* Program: InvestiGATOR Monitor
   Major Rev: 0
   Minor Rev: 0

   Author: Jon Mellott
   Date:   2/14/92F

   Description:
   This module contains the monitor program.
*/
/* --------- */
/* CONSTANTS */
/* --------- */

/* Logicals */
#define TRUE 1
#define FALSE 0

/* String constants. */
const char  cPrompt[]=">>";

/* ---- ----------- */
/* TYPE DEFINITIONS */
/* ---- ----------- */

#include "basetype.h"
```

```
/* -------- ---------- */
/* EXTERNAL REFERENCES */
/* -------- ---------- */

/* Reference to bit reversal table in module POSTINIT. */
extern BYTE BITREV[256];

/* Reference to bus error counter. */
extern DWORD dwBusErrorCount; /* This counts the number of bus errors. */

/* ----- ---------- */
/* MACRO DEFINITIONS */
/* ----- ---------- */

/* Min/Max macros. */
#define max(a,b)   (((a) > (b)) ? (a) : (b))
#define min(a,b)   (((a) < (b)) ? (a) : (b))

/* ------- --------- */
/* STATIC VARIABLES */
/* ------- --------- */


/* --------- -------- */
/* PROCEDURE DIVISION */
/* --------- -------- */

/* -------------------------------------
   Procedure: MonReadAddr(cBuffer,*iIndex,*dwAddress)

   Parameters:
   -----------
   cBuffer     -- Input string.
   *iIndex     -- Pointer to current position in string index.
   *dwAddress -- Address of variable to hold read address.
   -----------

   Description:
   This procedure takes the buffer starting at the index provided by
   iIndex and scans for a valid address which is then placed in the
   longword variable given by dwAddress.

   Returns: 0 for success, !0 for failure.
*/
int MonReadAddr(char *cBuffer, int *iIndex, DWORD *dwAddress)
{
  int   iError;

  while(cBuffer[*iIndex]!=0) ++*iIndex;
  while(cBuffer[*iIndex]==0) ++*iIndex;
  if ((BYTE)cBuffer[*iIndex]==0xFF) return 2;
  iError=sscanf(&cBuffer[*iIndex],"%lX",dwAddress);

  /* Done. */
  if (iError!=1) return 1;
  else return 0;
}

/* -------------------------------------
   Procedure: MonMemEdit(cBuffer)

   Parameters:
   -----------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure allows the user to edit memory contents by byte,
```

```
     word, or longword withing the given address range given by
     start-address and end-address.
     Returns: Nothing
*/
void MonMemEdit(char *cBuffer)
{
  DWORD dwStartAddress,dwEndAddress:
  DWORD      dwData;
  BYTE   *fpbyData;
  WORD   *fpwData;
  DWORD  *fpdwData;
  int        iError,iIndex,iSize:
  char       cTemp[20];

  /* Extract addresses. */
  iIndex=0;
  if (MonReadAddr(cBuffer,&iIndex,&dwStartAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  if (MonReadAddr(cBuffer,&iIndex,&dwEndAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  /* Get size [B|W|L]. */
  switch(cBuffer[2]) {
    case 'B':
      iSize=1;
      break;

    case 'W':
      iSize=2;
      break;

    case 'L':
      iSize=4;
      break;

    default:
      printf("\n***ERROR: Unable to determine size.");
      return ;
      break;
  }
  /* Walk through and edit memory. */
  for (; dwStartAddress<=dwEndAddress; dwStartAddress+=(DWORD)iSize) {
    switch(iSize) {
      case 1:
        fpbyData=(BYTE  *)dwStartAddress;
        printf("\n%81X [%2X]: ",dwStartAddress,*fpbyData);
        gets(cTemp);
        iError=sscanf(cTemp,"%1X",dwData);
        if (iError==1) *fpbyData=(BYTE)dwStartAddress;
        break;

      case 2:
        fpwData=(WORD  *)dwStartAddress;
        printf("\n%81X [%4X]: ",dwStartAddress,*fpwData);
        gets(cTemp);
        iError=sscanf(cTemp,"%1X",dwData);
        if (iError==1) *fpwData=(WORD)dwStartAddress;
        break;
```

```
      case 4:
        fpdwData=(DWORD  *)dwStartAddress;
        printf("\n%31X [%31X]: ",dwStartAddress,*fpdwData);
        gets(cTemp);
        iError=sscanf(cTemp,"%1X",dwData);
        if (iError==1) *fpdwData=(DWORD)dwStartAddress;
        break;
      default:
        printf("\n***ERROR: Something's really fubar.");
        return ;
        break;
    }
  }
  /* Done. */
  return;
}

/* -----------------------------------
   Procedure: MonMemMove(cBuffer)

   Parameters:
   ------------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure moves a block of memory defined by start-address and
   end-address to the area beginning at dest-address. Each of these
   arguments is given in the command line. The move is executed from
   start-address to end-address so overlapping moves *down* in memory
   will execute correctly, however, overlapping moves *up* in memory
   will not produce correct results.

   Returns: Nothing
*/
void MonMemMove(char *cBuffer)
{
  DWORD dwStartAddress,dwEndAddress,dwDestAddress;
  int   iIndex,iError;

  /* Extract addresses. */
  iIndex=0;
  if (MonReadAddr(cBuffer,&iIndex,&dwStartAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  if (MonReadAddr(cBuffer,&iIndex,&dwEndAddress)!=0) {
    printf("\n***ERROR: Unable to read end address.");
    return ;
  }
  if (MonReadAddr(cBuffer,&iIndex,&dwDestAddress)!=0) {
    printf("\n***ERROR: Unable to read destination address.");
    return ;
  }
  /* Validate address placement. */
  if (dwStartAddress>dwEndAddress) {
    printf("\n***ERROR: Start address comes after end address.");
    return ;
  }
  if (dwDestAddress>=dwStartAddress && dwDestAddress<=dwEndAddress) {
```

```
      printf("\n***ERROR: Destination address is in the block range.");
      return ;
   }
   printf("\nSorry, this function is not yet finished.");

   /* Done. */
   return ;
}
/* -----------------------------------
   Procedure: MonGo(cBuffer)

   Parameters:
   -----------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure begins program execution at the start-address given
   in the command line. Execution is started using a jump-to-subroutine,
   and the program may return control to the monitor by executing a
   return-from-subroutine instruction.

   Returns: Nothing
*/
void MonGo(char *cBuffer)
{
   int    iIndex,iError;
   DWORD  dwGoAddress;

   /* Extract the starting address. */
   iIndex=0;
   if (MonReadAddr(cBuffer,&iIndex,&dwGoAddress)!=0) {
      printf("\n***ERROR: Unable to read start address.");
      return ;
   }
   /* Validate address (check for word boundary). */
   if (dwGoAddress & (DWORD)1) {
      printf("\n***ERROR: Program execution must start on a word address.");
      return ;
   }
   printf("\nSorry, this function is not finished.");

   /* Done. */
   return ;
}
/* -----------------------------------
   Procedure: MonSMemLoad()

   Parameters:
   -----------
   -----------

   Description:
   This procedure takes S-record input from the standard input until
   a terminating S-record (S7, S8, or S9) is encountered. The S-record
   information is used to load memory.

   Returns: Nothing
*/
void MonSMemLoad()
{
   char   cBuffer[128];
   char   cTemp[160];
```

```
int     iRecLen,iIndex;
int     iTemp;
BOOL    bDone;
DWORD   dwStartAddress;
BYTE    byData;

/* Initialize. */
bDone=FALSE;

while(!bDone) {
  /* Read S-record. */
  gets(cBuffer);

  switch(cBuffer[1]) {
    case '0':
      break;

    case '1':
    case '2':
    case '3':
      /* Extract record length. */
      cTemp[0]=cBuffer[2];
      cTemp[1]=cBuffer[3];
      cTemp[2]=(char)0;
      sscanf(cTemp,"%X",&iRecLen);

      /* Extract starting address. */
      if (cBuffer[1]=='1') {
        for (iIndex=4; iIndex<8; ++iIndex) cTemp[iIndex-4]=cBuffer[iIndex];
        cTemp[4]=(char)0;
        iRecLen-=2;
      }
      else if (cBuffer[1]=='2') {
        for (iIndex=4; iIndex<10; ++iIndex) cTemp[iIndex-4]=cBuffer[iIndex];
        cTemp[6]=(char)0;
        iRecLen-=3;
      }
      else {
        for (iIndex=4; iIndex<12; ++iIndex) cTemp[iIndex-4]=cBuffer[iIndex];
        cTemp[8]=(char)0;
        iRecLen-=4;
      }
      sscanf(cTemp,"%lX",&dwStartAddress);
      printf("\n%81X: ",dwStartAddress);

      /* Extract the data. */
      while (iRecLen>1) {
        cTemp[0]=cBuffer[iIndex++];
        cTemp[1]=cBuffer[iIndex++];
        cTemp[2]=(char)0;
        sscanf(cTemp,"%X",&iTemp);
        byData=(BYTE)iTemp;

        printf("%02X ",(int)byData);

        /* Decrement counter. */
        --iRecLen;
      }
      break;

    case '7':
    case '8':
    case '9':
      printf("\nProcessing done.");
```

```
          bDone=TRUE;
          break;

      default:
          printf("\n***ERROR: Unable to determine record type.");
          break;
    }
  }
  /* Done. */
  return ;
}


/* --------------------------------------
   Procedure: MonSMemDump(cBuffer)

   Parameters:
   -----------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure dumps the memory from start-address to end-address
   in S-record format.

   Returns: Nothing
*/
void MonSMemDump(char *cBuffer)
{
  DWORD      dwStartAddress,dwEndAddress;
  BYTE    *fpbMem;
  BYTE       byChar;
  int        nBytes;
  int        iIndex;
  int        iCheckSum;

  /* Get start address. */
  iIndex=0;
  if (MonReadAddr(cBuffer,&iIndex,&dwStartAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  fpbMem=(BYTE *)dwStartAddress;

  /* Get end address. */
  if (MonReadAddr(cBuffer,&iIndex,&dwEndAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  /* Check that EndAddr comes after StartAddr. */
  if (dwEndAddress<dwStartAddress) {
    printf("\n***ERROR: Start address is greater than end address.");
    return ;
  }
  /* Print memory dump. */
  while(dwStartAddress<=dwEndAddress) {
    /* Compute number of bytes to dump. */
    nBytes=min((DWORD)(dwEndAddress-dwStartAddress+1),32);

    /* Print beginning of S-record. */
    if (dwStartAddress<(DWORD)0x10000)
      printf("\nS1%02X%04lX",nBytes+3,dwStartAddress);
    else if (dwStartAddress<(DWORD)0x1000000)
```

```
      printf("\nS2%02X%081X",nBytes+4,dwStartAddress);
    else printf("\nS3%02X%081X",nBytes+5,dwStartAddress);

    /* Dump data. */
    iCheckSum=(DWORD)0xFF & dwStartAddress +
             (DWORD)0xFF & dwStartAddress/0x100 +
             (DWORD)0xFF & dwStartAddress/0x10000 +
             (DWORD)0xFF & dwStartAddress/0x1000000;
    for (iIndex=0; iIndex<nBytes; ++iIndex) {
      byChar=(BYTE)*fpbMem;
      ++fpbMem;
      iCheckSum+=(int)byChar;
      printf("%02X",(BYTE)byChar);
    }
    /* Compute one's complement of checksum. */
    iCheckSum=0xFF & (~iCheckSum);
    printf("%02X",iCheckSum);

    /* Increment Start Address. */
    dwStartAddress+=nBytes;
  }
  /* Print junk terminator record. */
  printf("\nS9030000FC");

  return ;
}
/* -----------------------------------
   Procedure: MonAMemDump(cBuffer)

   Parameters:
   -----------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure dumps the memory from start-address to end-address
   in ASCII format.

   Returns: Nothing
*/
void MonAMemDump(char *cBuffer)
{
  DWORD      dwStartAddress,dwEndAddress;
  BYTE  *fpbMem;
  BYTE      byChar;
  int       iIndex;

  /* Get start address. */
  iIndex=0;
  if (MonReadAddr(cBuffer,&iIndex,&dwStartAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  fpbMem=(BYTE  *)dwStartAddress;

  if (MonReadAddr(cBuffer,&iIndex,&dwEndAddress)!=0) {
    dwEndAddress=dwStartAddress+(DWORD)63;
  }
  /* Check that EndAddr comes after StartAddr. */
  if (dwEndAddress<dwStartAddress) {
    printf("\n***ERROR: Start address is greater than end address.");
```

```
    return ;
  }
  /* Print memory dump. */
  while(dwStartAddress<=dwEndAddress) {
    iIndex=64;
    printf("\n%8lX: ",dwStartAddress);
    while (iIndex>0 && dwStartAddress<=dwEndAddress) {
      byChar=(BYTE)*fpbMem;
      ++fpbMem;
      if (byChar>=32 && byChar<=126) printf("%c",(char)byChar);
      else printf(".");
      ++dwStartAddress;
      --iIndex;
    }
  }
  return ;
}

/* ------------------------------------
   Procedure: MonMemDump(cBuffer)

   Parameters:
   -----------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure dumps the memory from start-address to end-address.

   Returns: Nothing
*/
void MonMemDump(char *cBuffer)
{
  DWORD     dwStartAddress,dwEndAddress;
  BYTE *fpbMem;
  int       iIndex;

  /* Get start address. */
  iIndex=0;
  if (MonReadAddr(cBuffer,&iIndex,&dwStartAddress)!=0) {
    printf("\n***ERROR: Unable to read start address.");
    return ;
  }
  fpbMem=(BYTE  *)dwStartAddress;

  switch (MonReadAddr(cBuffer,&iIndex,&dwEndAddress)) {
    case 1:
      printf("\n***ERROR: Unable to read start address.");
      return ;
      break;
    case 2:
      dwEndAddress=dwStartAddress+(DWORD)15;
      break;
    default:
      break;
  }
  /* Check that EndAddr comes after StartAddr. */
  if (dwEndAddress<dwStartAddress) {
    printf("\n***ERROR: Start address is greater than end address.");
    return ;
  }
  /* Print memory dump. *
```

```
  while(dwStartAddress<=dwEndAddress) {
    iIndex=16;
    printf("\n%81X: ",dwStartAddress);
    while (iIndex>0 && dwStartAddress<=dwEndAddress) {
      printf("%2X ",*fpbMem);
      ++fpbMem;
      ++dwStartAddress;
      --iIndex;
    }
  }
  return ;
}
/* ------------------------------------
   Procedure: MonParse(cBuffer)

   Parameters:
   -----------
   cBuffer -- Input string.
   -----------

   Description:
   This procedure places NUL characters after each word in the buffer.
   The end of the string NUL is followed by a 0xFF to indicate the
   end of strings.

   Returns: Nothing
*/
void MonParse(char *cBuffer)
{
  BOOL bDone;
  int  iIndex;
  /* Initialize. */
  bDone=FALSE;
  iIndex=0;
  /* Begin main loop. */
  while (!bDone) {
    switch(cBuffer[iIndex]) {
      case 0:
        cBuffer[iIndex+1]=(char)0xFF;
        bDone=TRUE;
        break;

      case ' ':
        cBuffer[iIndex]=(char)0;
        break;

      default:
        break;
    }
    ++iIndex;
  }
  /* Done. */
  return ;
}

/* ------------- ----  --------------------- ----
   Procedure: Monitor()

   Parameters:
   ------------
   ------------
```

208

```c
   Description:
   This procedure is the monitor program.

   Returns: Nothing
*/
void Monitor()
{
  BOOL bExit;
  char cBuffer[30];

  /* Print banner. */
  printf("\nInvestiGATOR Monitor Program");
  printf("\nversion 0.0");
  printf("\n2/14/92");
  printf("\nJ. Mellott");
  printf("\nHigh Speed Digital Architecture Laboratory\n");

  /* Initialize monitor. */
  bExit=FALSE;

  /* Main monitor execution loop. */
  while(!bExit) {
    printf("\n%s",cPrompt);
    gets(cBuffer);

    /* Parse the string. */
    { /* Convert to uppercase. */
      int i;
      i=0;
      while(cBuffer[i]!=0) {
        cBuffer[i]=(char)toupper((int)cBuffer[i]);
        ++i;
      }
    }

    /* 'Parse' the character buffer so it can be treated as a string of
       ASCIIZ strings. */
    MonParse(cBuffer);

    switch(cBuffer[0]) {

      case 'A':  /* ASCII memory dump. */
        MonAMemDump(cBuffer);
        break;

      case 'B':  /* Bus error counter print. */
        /* Print number of bus errors which have occured. */
        printf("\nThere have been %1X bus errors.",dwBusErrorCount);
        break;

      case 'D':  /* Memory dump. */
        MonMemDump(cBuffer);
        break;

      case 'G':  /* Begin program execution. */
        MonGo(cBuffer);
        break;

      case 'L':  /* Load S-records. */
        MonSMemLoad();
        break;

      case 'M':  /* Move memory block. */
        switch(cBuffer[1]) {
          case 'M':
            MonMemMove(cBuffer);
```

```
            break;
          case 'V':
            MonMemMove(cBuffer);
            break;
          default:
            printf("\n***ERROR: Unknown command.");
            break;
        }
        break;
      case 'S':   /* Save S-records. */
        MonSMemDump(cBuffer);
        break;
      case 'V':   /* Print version number(s). */
        printf("\nInvestiGATOR Monitor version 0.0");
        break;
      case 'Q':  /* Exit monitor. */
        printf("\nExiting monitor...");
        bExit=TRUE;
        break;
      default:
        printf("\n***ERROR: Unknown command.");
        break;
    }
  }
  /* Monitor ends... */
  return ;
}

/* ---- ------- */
/* MAIN PROGRAM */
/* ---- ------- */
#if TESTVERSION
main()
{
  /* Call monitor program. */
  Monitor();
}
#endif
```

D.7 Serial I/O Code: ESCC.C

```
/*
  InvestiGATOR Backplane Firmware for Z85C30 ESCC
  ------------ --------- -------- --- ------ ----

  Major Rev: 0
  Minor Rev: 0
  Date: 2/6/92

  Author: Jon Mellott

  High Speed Digital Architecture Laboratory
  University of Florida
  405 CSE Building
  Gainesville, FL  32611
```

210

```
   This module contains initialization and interface code
   for the Z85C30 ESCC.
*/

/* ------- ------- */
/* INCLUDE SECTION */
/* ------- ------- */

/* ----- ----------- */
/* MACRO DEFINITIONS */
/* ----- ----------- */
#define PUT8(addr,val)  (*(unsigned char *)(addr))=val
#define GET8(addr)      (*(unsigned char *)(addr))

/* --------- ------- */
/* CONSTANTS SECTION */
/* --------- ------- */

/* Buffered/non-buffered ESCC I/O flag (nobuf=1, buf=0). */
#define ESCCNBUF 0

/* Logicals */
#define TRUE    1
#define FALSE   0

/* ASCII Constants */
#define XON     0x11 /* ^Q */
#define XOFF    0x13 /* ^S */

/* Serial Port Definitions */
#define ESCC_CACR   0x00020802
#define ESCC_CADR   0x00020803
#define ESCC_CBCR   0x00020801
#define ESCC_CBDR   0x00020800

/* ESCC register masks. */
#define ESCC_RXAV   0x80
#define ESCC_TXEM   0x20

/* ESCC buffer length definition. Note: this definition must match that
   found in INVESTIO.INC. */
#define ESCCBUFL    128

/* ESCC buffer index mask definition. This is tied to ESCCBUFL. */
#define ESCCBUFW    127

/* ESCC character buffer fill threshold for XOFF.  Note: this definiton
   must match that found in INVESTIO.INC. */
#define ESCCXOFT    96

/* ESCC XON/XOFF request flag definitions. These are also defined in
   INVESTIO.INC */
#define NOXREQ      (BYTE)0     /* No XON/XOFF request, state=XON.  */
#define XOFFREQ     (BYTE)1     /* XOFF request flag, state=XON.    */
#define XONREQ      (BYTE)2     /* XON request flag, state=XOFF.    */
#define RCVXOFF     (BYTE)3     /* No XON/XOFF request, state=XOFF. */

/* ---- ----------- */
/* TYPE DEFINITIONS */
/* ---- ----------- */

#include "basetype.h"

/* --------- ----------- */
/* EXTERNAL REFERENCES */
/* --------- ----------- */
```

```
/* Reference to bit reversal table in module POSTINIT. */
extern BYTE BITREV[256];
/* Serial port statics: defined in ISR.M68. */
extern BYTE byAReqXonXoff;              /* ESCC port A xmit XON/XOFF req flag. */
extern BYTE byBReqXonXoff;              /* ESCC port B xmit XON/XOFF req flag. */
volatile extern BOOL bATxXoff;          /* ESCC port A xmitr XON/XOFF flag.    */
volatile extern BOOL bBTxXoff;          /* ESCC port B xmitr XON/XOFF flag.    */
extern char *cpARxBuff;                 /* ESCC Port A receive buffer.         */
extern char *cpATxBuff;                 /* ESCC Port B transmit buffer.        */
extern char *cpBRxBuff;                 /* ESCC Port A receive buffer.         */
extern char *cpBTxBuff;                 /* ESCC Port B transmit buffer.        */
extern char cARxBuffer[];               /* ESCC Port A receive buffer array.   */
extern char cATxBuffer[];               /* ESCC Port A transmit buffer array.  */
extern char cBRxBuffer[];               /* ESCC Port B receive buffer array.   */
extern char cBTxBuffer[];               /* ESCC Port B transmit buffer array.  */
volatile extern int nARxHead;           /* ESCC Port A receive buffer head.    */
volatile extern int nBRxHead;           /* ESCC Port B receive buffer head.    */
volatile extern int nATxHead;           /* ESCC Port A transmit buffer head.   */
volatile extern int nBTxHead;           /* ESCC Port A transmit buffer head.   */
volatile extern int nARxTail;           /* ESCC Port A receive buffer tail.    */
volatile extern int nBRxTail;           /* ESCC Port B receive buffer tail.    */
volatile extern int nATxTail;           /* ESCC Port A transmit buffer tail.   */
volatile extern int nBTxTail;           /* ESCC Port B transmit buffer tail.   */

/* Set IPL procedure defined in ISR.M68. */
extern void SetIPL(int);

/* ------- ------------- */
/* STATIC DECLARATIONS */
/* ------- ------------- */


/* ---------- -------- */
/* PROCEDURE DIVISION */
/* ---------- -------- */

/* -----------------------------------------------------------------

   SerialPortInit()

   Parameters: None

   This procedure causes the Z85C30 serial port to be initialized.

   Returns: Nothing

*/

void SerialPortInit()
{
  /* Initialize A port. */

  /* Initialize B port. */

  /* Initialize buffer control variables. */
  bATxXoff=FALSE;                         /* Set transmit XOFF flag false. */
  bBTxXoff=FALSE;

  byAReqXonXoff=NOXREQ;                   /* Set request XON/XOFF to noreq */
  byBReqXonXoff=NOXREQ;

  cpARxBuff=cARxBuffer;                   /* Initialize buffer pointers.   */
  cpATxBuff=cATxBuffer;
  cpBRxBuff=cBRxBuffer;
  cpBTxBuff=cBTxBuffer;

  nARxHead=0;                             /* Initialize buffer indeces. */
  nARxTail=0;
```

```
    nATxHead=0;
    nARxTail=0;
    nBRxHead=0;
    nBRxTail=0;
    nBTxHead=0;
    nBTxTail=0;
    /* Done. */
    return ;
}
/* -------------------------------------------------------------------
    _s_getch(iAddr)

    Parameters:
    -----------
    iAddr:    port address
    -----------

    This procedure gets a character from the buffer or ESCC.

    Returns: A character.
*/
#if ESCCNBUF
/* The following is the non-buffered version of the _s_getch procedure. */
_s_getch(int iAddr)
{
  register char cChar:

  /* Wait for character in port A Rx buffer. */
  while(!_s_kbhit(ESCC_CACR));

  cChar=BITREV[(BYTE)GETB(ESCC_CADR)];

  /* Done. */
  return (cChar):
}
#else
/* The following is the buffered version of the _s_getch procedure. */
_s_getch(int iAddr)
{
  register char cChar;
  /* Check for character in buffer A. */
  while (TRUE) {
    if (nARxHead != nARxTail) {
      /* Get character from buffer. */
      cChar=cARxBuffer[nARxTail];

      /* Increment tail index and perform wrap-around. */
      ++nARxTail:
      nARxTail=nARxTail & ESCCBUFW;

      /* If the receiver state is XOFF and buffer length is less
         than threshold, and there is no pending XON/XOFF request
         then request transmission of XON. */
      if (byAReqXonXoff=RCVXOFF) { /* If XOFF... */
        if (((nARxHead-nARxTail) % ESCCBUFL) < ESCCXOFT) {
          byAReqXonXoff=XONREQ;
        }
      }

      /* Done. */
      return (cChar) ;
    }
  }
```

```
}
#endif

/* -----------------------------------------------------------------
   getch()

   Parameters: None

   This procedure gets a character from the buffer or ESCC and is a
   wrapper for _s_getch().

   Returns: A character.
*/
int getch()
{
  _s_getch(ESCC_CADR);
}

/* -----------------------------------------------------------------
   _s_kbhit(iAddr)

   Parameters:
   -----------
   iAddr:     port address
   -----------

   This procedure tests for a character on the ESCC.

   Returns: TRUE if there is a character, FALSE otherwise.
*/
#if ESCCNBUF
/* The following procedure is the non-buffered version of _s_kbhit. */
_s_kbhit(int iAddr)
{
  BYTE cChar;
  cChar=BITREV[(BYTE) GET8(ESCC_CACR)]; /* Read register zero. */
  if (cChar & ESCC_RXAV) return (TRUE); /* Yes, there is a character. */

  /* No, there is not a character. */
  return (FALSE);
}
#else
/* The following procedure is the buffered version of _s_kbhit. */
_s_kbhit(int iAddr)
{
  /* Check receive buffer for characters. */
  if (nARxHead==nARxTail) return (TRUE);     /* Character in buffer. */
  else return(FALSE) ;                       /* No character in buffer. */
}
#endif
/* -----------------------------------------------------------------
   kbhit()

   Parameters: None

   This procedure tests for a character on the ESCC and is a wrapper
   for _s_kbhit.

   Returns: TRUE if character exists on ESCC, FALSE otherwise
*/
int kbhit()
{
  return(_s_kbhit(ESCC_CADR));
```

```
        :

}
/* ----------------------------------------------------------------------
    _s_putch(iAddr,cData)

    Parameters:
    -----------
    iAddr:      port address
    cData:      Character.
    -----------

    This procedure transmits a character on the ESCC.

    Returns: Nothing.
*/
#if ESCCNBUF
/* The following rpocedure is the non-buffered version of _s_putch. */
_s_putch(int iAddr, int iData)
{
  char cTmp;

  /* Wait until Tx buffer empty. */
  while(!(BITREV[(BYTE)GET8(ESCC_CACR)] & ESCC_TXEM));

  /* Transmit the character. */
  PUT8(ESCC_CADR,BITREV[(BYTE)(0xFF & iData)]);

  /* Done. */
  return :
}
#else
/* The following procedure is the buffered version of _s_putch. */
_s_putch(int iAddr, int iData)
{
  char cTmp;
  LONG lThreshold;

  /* Begin section which should not be interrupted by the ESCC ISR. */
  SetIPL(0x700);

  if (nATxHead==nATxTail && !bATxXoff) {
    /* Put character to ESCC. */
    for (;;) {
      PUT8(ESCC_CACR,0x00); /* Read register zero. */
      if (BITREV[(BYTE) GET8(ESCC_CACR)] & ESCC_TXEM) { /* TX buffer empty. */
        PUTS(ESCC_CADR,BITREV[(BYTE)(0xFF & iData)]); /* Put the character. */
        break:
      }
    }
  }
  else {
    /* Check for Tx buffer full, and wait if full. */
    while (((nATxHead-nATxTail) % ESCCBUFL) > ESCCXOFT) ;

    /* Add character to buffer. */
    cATxBuffer[nATxHead]=(char)(0xFF & iData);

    /* Increment head counter and perform wrap-around. */
    ++nATxHead:
    nATxHead=nATxHead & ESCCBUFW;
  }

  /* Restore IPL level. */
  SetIPL(0x0000);

  /* Done. */
```

```
  return ;
}
#endif
/* ------------------------------------------------------------
   putch(cData)

   Parameters:
   -----------
   iData:      Character.
   -----------

   This procedure transmits a character on the ESCC and is a wrapper
   for _s_putch.

   Returns: Zero.
*/
int putch(int iData)
{
  return(_s_putch(ESCC_CADR,iData));
}

/* ------------------------------------------------------------
   _s_exit()

   Parameters: None

   This procedure is the exit to monitor procedure. There is no
   monitor so the program puts itself in a loop.

   Returns: This procedure never returns.
*/
void _s_exit()
{
  for (;;) ;
}

/* ------------------------------------------------------------
   _s_clkinit()

   Parameters: None

   This procedure is supposed to initialize the system clock. The
   InvestiGATOR lacks a clock so this is a null call.

   Returns: Nothing.
*/
void _s_clkinit()
{
  return;
}

/* ------------------------------------------------------------
   _s_time()

   Parameters: None

   This procedure is supposed to return the current time. The
   InvestiGATOR lacks a clock so this is a null call.

   Returns: Time=0.
*/
long _s_time()
{
  return 0;
}

/* ------------------------------------------------------------
   _s_init_port()
```

```
   Parameters: None

   This procedure is a wrapper for SerialPortInit.

   Returns: Nothing.
*/
_s_init_port()
{
  /* Initialize serial port. */
  SerialPortInit;

  /* Done. */
  return;
}
```

D.8 SCSI I/O Code: SBIC.C

```
/*
  InvestiGATOR Backplane Firmware for Am33C93A SBIC
  ------------ --------- -------- --- -------- ----
  Major Rev: 0
  Minor Rev: 0
  Date: 2/6/92

  Author: Jon Mellott

  High Speed Digital Architecture Laboratory
  University of Florida
  405 CSE Building
  Gainesville, FL  32611

  This code constitutes the SBIC firmware.
*/
/* ------- ------- */
/* INCLUDE SECTION */
/* ------- ------- */

/* ----- ----------- */
/* MACRO DEFINITIONS */
/* ----- ----------- */
#define PUT8(addr,val)  (*(unsigned char *)(addr))=val
#define GET8(addr)      (*(unsigned char *)(addr))

/* --------- ------- */
/* CONSTANTS SECTION */
/* --------- ------- */

/* Logicals */
#define TRUE     1
#define FALSE    0

/* ASCII Constants */
#define XON      0x11 /* ^Q */
#define XOFF     0x13 /* ^S */

/* SBIC Port Definitions */
#define SCSI_PORT_ID 5
#define SBIC_ADDR      0x00020000
#define SBIC_OWNID     SBIC_ADDR
#define SBIC_CTRL_REG  SBIC_ADDR+1
#define SBIC_TIMEOUT   SBIC_ADDR+2
#define SBIC_TOT_SECT  SBIC_ADDR+3
```

```
#define SBIC_TOT_HEAD SBIC_ADDR+4
#define SBIC_CYLN_MSB SBIC_ADDR+5
#define SBIC_CYLN_LSB SBIC_ADDR+6
#define SBIC_LOGADDRM SBIC_ADDR+7
#define SBIC_LOGADDR2 SBIC_ADDR+8
#define SBIC_LOGADDR3 SBIC_ADDR+9
#define SBIC_LOGADDRL SBIC_ADDR+10
#define SBIC_SECTNUMR SBIC_ADDR+11
#define SBIC_HEADNUMR SBIC_ADDR+12
#define SBIC_CYLNUM_H SBIC_ADDR+13
#define SBIC_CYLNUM_L SBIC_ADDR+14
#define SBIC_TARG_LUN SBIC_ADDR+15
#define SBIC_CMD_PHAS SBIC_ADDR+16
#define SBIC_SYNCH_TR SBIC_ADDR+17
#define SBIC_TCNTR_M  SBIC_ADDR+18
#define SBIC_TCNTR_2  SBIC_ADDR+19
#define SBIC_TCNTR_L  SBIC_ADDR+20
#define SBIC_DEST_ID  SBIC_ADDR+21
#define SBIC_SRC_ID   SBIC_ADDR+22
#define SBIC_SCSI_ST  SBIC_ADDR+23
#define SBIC_CMD_REG  SBIC_ADDR+24
#define SBIC_DATAREG  SBIC_ADDR+25

/* SBIC Commands. */
#define SBIC_RESET      0x00
#define SBIC_ABORT      0x01
#define SBIC_ASSATN     0x02
#define SBIC_NEGATN     0x03
#define SBIC_DISCON     0x04
#define SBIC_RESEL      0x05
#define SBIC_SELWATN    0x06
#define SBIC_SELOATN    0x07
#define SBIC_SELWATNT   0x08
#define SBIC_SELOATNT   0x09
#define SBIC_RESELRD    0x0A
#define SBIC_RESELSD    0x0B
#define SBIC_WFSELRCV   0x0C
#define SBIC_SSTATCMP   0x0D
#define SBIC_SENDDISC   0x0E
#define SBIC_SETIDI     0x0F
#define SBIC_RECCMD     0x10
#define SBIC_RECDATA    0x11
#define SBIC_RECMESO    0x12
#define SBIC_RECUIO     0x13
#define SBIC_SENDSTAT   0x14
#define SBIC_SENDDATA   0x15
#define SBIC_SENDMESI   0x16
#define SBIC_SENDUII    0x17
#define SBIC_TRANSADR   0x18
#define SBIC_TRANSINF   0x20

/* SBIC Interrupt Status Codes. */
/* These codes are defined for the SCSI Status Register. */
#define SBIC_IRQ_RESET    0x00 /* SBIC Reset, no advanced features. */
#define SBIC_IRQ_RESETA   0x01 /* SBIC Reset, adv. features enabled. */

#define SBIC_IRQ_RESELT   0x10 /* Resel. comp. suc., con. as target. */
#define SBIC_IRQ_SELI     0x11 /* Sel. comp. suc., con. as initiator. */
#define SBIC_IRQ_RECNA    0x13 /* Cmd. comp. suc., ATN not asserted. */
#define SBIC_IRQ_RECA     0x14 /* Cmd. comp. suc., ATN asserted. */
#define SBIC_IRQ_ADRTRNS  0x15 /* Addr. translation comp. successfully. */
#define SBIC_IRQ_SELT     0x16 /* Select and transfer comp. suc. */
#define SBIC_IRQ_SUCHCI   0x18 /* HCI Transfer succeeded. */
```

```
#define SBIC_IRQ_TIPA          0x20 /* Trans info cmd paused w/ ACK */
#define SBIC_IRQ_SDPST         0x21 /* Save-data-point.r mes. rec. during S&T */
#define SBIC_IRQ_SELAB         0x22 /* Select/Reselect command aborted. */
#define SBIC_IRQ_RECABNA       0x23 /* Rec/Send err/abort, ATN not asserted. */
#define SBIC_IRQ_RECABA        0x24 /* Rec/Send err/abort, ATN asserted. */
#define SBIC_IRQ_RESELF        0x27 /* Resel dur S&T, !=target. */
#define SBIC_IRQ_TAMCI         0x28 /* Transfer aborted, new phase=MCI. */

#define SBIC_IRQ_INVCMD        0x40 /* Invalid command issued. */
#define SBIC_IRQ_UNEXDISC      0x41 /* Unexpected disconnect by target. */
#define SBIC_IRQ_SELTIMO       0x42 /* Timeout during S&T, SBIC disconnected. */
#define SBIC_IRQ_PENA          0x43 /* Parity error, ATN not asserted. */
#define SBIC_IRQ_PEA           0x44 /* Parity error, ATN asserted. */
#define SBIC_IRQ_LAEDB         0x45 /* Logical address exceded disk boundary. */
#define SBIC_IRQ_NARESF        0x46 /* Resel dur S&T, !=target, not adv mode */
#define SBIC_IRQ_INCSTAT       0x47 /* Incorrect status byte rec. dur. S&T. */
#define SBIC_IRQ_UNEXPMCI      0x48 /* Unexpected phase req, req=MCI. */

#define SBIC_IRQ_RESNOID       0x80 /* Resel, con. as initiator, no ID mes. */
#define SBIC_IRQ_RESADV        0x81 /* Resel in advanced mode. */
#define SBIC_IRQ_SELNA         0x82 /* Selected as target, ATN not asserted. */
#define SBIC_IRQ_SELA          0x83 /* Selected as target, ATN asserted. */
#define SBIC_IRQ_ATN           0x84 /* ATN signal asserted. */
#define SBIC_IRQ_DISCON        0x85 /* A disconnect has occured. */
#define SBIC_IRQ_WSRP          0x87 /* A wait for sel & rec has paused. */
#define SBIC_IRQ_REQMCI        0x88 /* The REQ has been asserted, req=MCI. */

/* SBIC Interrupt Status MCI Codes. */
/* MCI codes are masked with SBIC interrupt status codes of type MCI
   to produce the full SBIC status code. */
#define SBIC_MCI_DO            0x0  /* Data out phase. */
#define SBIC_MCI_DI            0x1  /* Data in phase. */
#define SBIC_MCI_CMD           0x2  /* Command phase. */
#define SBIC_MCI_STAT          0x3  /* Status phase. */
#define SBIC_MCI_UIO           0x4  /* Unspecified info out phase. */
#define SBIC_MCI_UII           0x5  /* Unspecified info in phase. */
#define SBIC_MCI_MO            0x6  /* Message out phase. */
#define SBIC_MCI_MI            0x7  /* Message in phase. */

/* SCSI Commands */
/*   Group 0. */
#define TEST_UNIT_READY        0x00
#define REQUEST_SENSE          0x03
#define RECEIVE                0x08
#define SEND                   0x0A
#define INQUIRY                0x12
#define RECV_DIAG_RESULTS      0x1C
#define SEND_DIAGNOSTIC        0x1D

/* SCSI Message Code Values */
#define SCMS_CMDCMP            0x00
#define SCMS_EXTMES            0x01
#define SCMS_SAVDP             0x02
#define SCMS_RESPTR            0x03
#define SCMS_DISCON            0x04
#define SCMS_INIDERR           0x05
#define SCMS_ABORT             0x06
#define SCMS_MESREJ            0x07
#define SCMS_NOOP              0x08
#define SCMS_MESPE             0x09
```

```
#define SCHS_LCMDCMP        0x0A
#define SCHS_LCMDCMPF       0x0B
#define SCHS_BUSDRES        0x0C
#define SCHS_IDENTIFY       0x80

/* SCSI Status Byte Code Values */
#define STAT_GOOD           0x00
#define STAT_CHKC           0x04
#define STAT_CDMG           0x06
#define STAT_BUSY           0x08
#define STAT_IMDG           0x10
#define STAT_ICMG           0x14
#define STAT_RESC           0x18


/* ---- ----------- */
/* TYPE DEFINITIONS */
/* ---- ----------- */

#include "basetype.h"

/* SCSI Command Frame Type Definitions. */

typedef struct _SCSISixByteFrame {
  BYTE    ucOpcode;
  BYTE    ucLUN_LBA;
  BYTE    ucLBA,ucLBA_LSB;
  BYTE    ucControl;
} SCSISixByteFrame;

typedef struct _SCSITenByteFrame {
  BYTE    ucOpcode;
  BYTE    ucLUN;
  DWORD   ulLBA;
  BYTE    ucReserved;
  WORD    uiXferLen;
  BYTE    ucControl;
} SCSITenByteFrame;

typedef struct _SCSITwelveByteFrame {
  BYTE    ucOpcode;
  BYTE    ucLUN;
  WORD    ulLBA;
  BYTE    ucReserved1,ucReserved2,ucReserved3;
  WORD    uiXferLen;
  BYTE    ucControl;
} SCSITwelveByteFrame;

/* -------- ---------- */
/* EXTERNAL REFERENCES */
/* -------- ---------- */

extern union {
  SCSISixByteFrame      Six;
  SCSITenByteFrame      Ten;
  SCSITwelveByteFrame   Twelve;
} SCSCmdFrame;

/* ------- ------------ */
/* STATIC DECLARATIONS */
/* ------- ------------ */

/* --------- --------- */
/* PROCEDURE DIVISION */
/* --------- --------- */
```

```
/* ---------------------------------------------------------------
   SCSIPortInit(bIntEnable)

   Parameters:
   -----------
   bIntEnable -- TRUE -> Interrupts enabled.
                 FALSE -> Interrupts disabled.

   This procedure initializes the SCSI port.

   Returns: Nothing
*/
void SCSIPortInit(BOOL bIntEnable)
{
  /* Reset the SBIC and set the SCSI ID of the InvestiGATOR.
     Enable advanced features. */
  PUT8(SBIC_OWNID,SCSI_PORT_ID & 0x84);

  if (bIntEnable) {
    /* Setup the SBIC for polled I/O mode, no halt on parity fault, ATN*,
       and disable disconnect interrupts. */
    PUT8(SBIC_CTRL_REG,0x00);
  }
  else {
    /* Setup the SBIC for polled I/O mode, no halt on parity fault, ATN*,
       and disable disconnect interrupts. */
    PUT8(SBIC_CTRL_REG,0x00);
  }

  /* Disable timeout for select/reselect commands. */
  PUT8(SBIC_TIMEOUT,0x00);

  /* Done. */
  return ;
}
```

D.9   Interrupt Service Routines: ISR.M68

```
; Description: Exception Vector Table and Interrupt Service Routines for
; ------------ Backplane Firmware
;
; Major Rev: 0
; Minor Rev: 0
; Date: 2/6/92
;
; Author: Jon Mellott
;
; High Speed Digital Architecture Laboratory
; University of Florida
; 405 CSE Building
; Gainesville, FL  32611
; --------------- --------
; IDENTIFICATION DIVISION
; --------------- --------
ISR      IDNT 0,1        ; Module ISR, Major rev=0, Minor rev=0
         SECTION 0       ; Place in section zero (ROM)
; ----------- --------
; ENVIRONMENT DIVISION
; ----------- --------
         OPT      HOMEX,P=68030
```

```
;  -------- ----------
;  EXTERNAL REFERENCES
;  -------- ----------
        XREF  .CENTRY    ; Startup module reference.
        XREF  POSTINIT   ; POST Code entry point.
        XREF  BITREV     ; Bit reversal table.

;  -------
;  EXPORTS
;  -------
        XDEF  cARxBuffer      ; ESCC Port A receive buffer.
        XDEF  cATxBuffer      ; ESCC Port B transmit buffer.
        XDEF  cBRxBuffer      ; ESCC Port A receive buffer.
        XDEF  cBTxBuffer      ; ESCC Port B transmit buffer.

        XDEF  cpARxBuff       ; Pointer to port A Rx buffer.
        XDEF  nARxHead        ; Port A head index.
        XDEF  nARxTail        ; Port A tail index.
        XDEF  cpATxBuff       ; Pointer to port A Rx buffer.
        XDEF  nATxHead        ; Port A head index.
        XDEF  nATxTail        ; Port A tail index.
        XDEF  bATxXoff        ; Port A transmit disable boolean flag.
        XDEF  byAReqXonXoff   ; Port A transmit XON/XOFF request flag.

        XDEF  cpBRxBuff       ; Pointer to port A Rx buffer.
        XDEF  nBRxHead        ; Port A head index.
        XDEF  nBRxTail        ; Port A tail index.
        XDEF  cpBTxBuff       ; Pointer to port A Rx buffer.
        XDEF  nBTxHead        ; Port A head index.
        XDEF  nBTxTail        ; Port A tail index.
        XDEF  bBTxXoff        ; Port A transmit disable boolean flag.
        XDEF  byBReqXonXoff   ; Port A transmit XON/XOFF request flag.

        XDEF  bySCSSourceID   ; SCSI Source ID value.
        XDEF  bSCSSIDValid    ; SCSI Source ID valid flag.
        XDEF  SCSCmdFrame     ; SCSI Command frame storage.

        XDEF  dwBusErrorCount ; Bus error counter.

        XDEF  SetIPL          ; SetIPL procedure.

;  -------
;  EQUATES
;  -------
; Base location of RAM.
RAM_BASE  EQU       $100000

; InvestiGATOR I/O port location and misc. constants.
          INCLUDE   INVESTIO.INC

;  --------- ------ ----- -----------
;  Exception Vector Table Definition.
;  --------- ------ ----- -----------
          SECTION   1       ; Section 1 is the text section.
          ORG       $0      ; Ensure that EVT is located @ $0.
EXCEPTION_TABLE:
; Reset Initial Interrupt Stack Pointer
          DC.L      $1FFFFF
; Reset Initial Program Counter
          DC.L      POSTINIT
; Bus Error Exception
```

```
                    :

                DC.L        BERR_INTERRUPT
; Address Error Exception
                DC.L        NULL_INTERRUPT
; Illegal Instruction
                DC.L        NULL_INTERRUPT
; Zero Divide Trap
                DC.L        NULL_INTERRUPT
; CHK, CHK2 Instruction Trap
                DC.L        NULL_INTERRUPT
; cpTRAPcc, TRAPcc, TRAPV Instructions
                DC.L        NULL_INTERRUPT
; Privledge Violation
                DC.L        NULL_INTERRUPT
; Trace
                DC.L        NULL_INTERRUPT
; Line 1010 Emulator
                DC.L        NULL_INTERRUPT
; Line 1111 Emulator
                DC.L        NULL_INTERRUPT
; (Reserved)
                DC.L        $0
; Coprocessor Protocol Violation
                DC.L        NULL_INTERRUPT
; Format Error
                DC.L        NULL_INTERRUPT
; Unitialized Interrupt
                DC.L        NULL_INTERRUPT
; (Reserved)
                DC.L        $0,$0,$0,$0,$0,$0,$0,$0
; Spurious Interrupt
                DC.L        NULL_INTERRUPT
; Level 1 Interrupt Autovector
                DC.L        ARRAY_INTERRUPT
; Level 2 Interrupt Autovector
                DC.L        IOBUS_INTERRUPT
; Level 3 Interrupt Autovector
                DC.L        NULL_INTERRUPT
; Level 4 Interrupt Autovector
                DC.L        SIO_INTERRUPT
; Level 5 Interrupt Autovector
                DC.L        NULL_INTERRUPT
; Level 6 Interrupt Autovector
                DC.L        SCSI_INTERRUPT
; Level 7 Interrupt Autovector
                DC.L        NULL_INTERRUPT
; TRAP #0 Instruction Vector
                DC.L        NULL_INTERRUPT
; TRAP #1 Instruction Vector
                DC.L        NULL_INTERRUPT
; TRAP #2 Instruction Vector
                DC.L        NULL_INTERRUPT
; TRAP #3 Instruction Vector
```

```
              DC.L      NULL_INTERRUPT
; TRAP #4 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #5 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #6 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #7 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #8 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #9 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #10 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #11 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #12 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #13 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #14 Instruction Vector
              DC.L      NULL_INTERRUPT
; TRAP #15 Instruction Vector
              DC.L      NULL_INTERRUPT
; FPCP Branch or Set on Unordered Condition
              DC.L      NULL_INTERRUPT
; FPCP Inexact Result
              DC.L      NULL_INTERRUPT
; FPCP Divide by Zero
              DC.L      NULL_INTERRUPT
; FPCP Underflow
              DC.L      NULL_INTERRUPT
; FPCP Operand Error
              DC.L      NULL_INTERRUPT
; FPCP Overflow
              DC.L      NULL_INTERRUPT
; FPCP Signaling NAN
              DC.L      NULL_INTERRUPT
; (Reserved)
              DC.L      $0
; MMU Configuration Error
              DC.L      NULL_INTERRUPT
; Defined for MC68851 not used by MC68030
              DC.L      NULL_INTERRUPT
; Defined for MC68851 not used by MC68030
              DC.L      NULL_INTERRUPT
; Unassigned, reserved
              DC.L      NULL_INTERRUPT
              DC.L      NULL_INTERRUPT
              DC.L      NULL_INTERRUPT
              DC.L      NULL_INTERRUPT
              DC.L      NULL_INTERRUPT
```

```
; User defined vectors
         DCB.L    192,NULL_INTERRUPT
;------------------------------------------------------------------
; --------- --------
; INTERRUPT HANDLERS
; --------- --------
         SECTION  0 ; Section 0 is the code section.

NULL_INTERRUPT:   ; This handler used by undefined exceptions.
         RTE     ; Return to execution.

;------------------------------------------------------------------
; ARRAY INTERRUPT HANDLER
;
; Description:
; Service of this interrupt is target system dependent.

ARRAY_INTERRUPT:
         RTE

;------------------------------------------------------------------
; I/O BUS INTERRUPT HANDLER
;
; Description:
; Service of this interrupt is target system dependent.

IOBUS_INTERRUPT:
         RTE

;------------------------------------------------------------------
; SERIAL I/O INTERRUPT HANDLER
;
; Description:
; This interrupt services the serial port.
; The processing steps are as follows:
;
; 1. Determine the source of the interrupt. Possible sources are the
;    following:
;    Receiver Interrupt sources:
;      INT on 1st Rx char or special condition
;      INT on all Rx char or special condition (***)
;      Rx INT on special condition only
;      Parity
;    Transmit Buffer Empty (***)
;    External/Status Interrupt Sources:
;      Zero count
;      DCD
;      SYNC/HUNT
;      CTS
;      Tx Underrun/EOM
;      Break/abort
;
;    The actual interrupt conditions are limitted to those indicated by
;    (***) above. Interrupt source may be determined by examining RR3.
;
; 2. Perform interrupt service.
;    If Tx buffer empty, check buffer and transmit next byte.
;    If Rx char, get char and place in buffer.
;    If Rx special condition, then handle.

SIO_INTERRUPT:
         ; Save registers.
         MOVEM.L   D0-D7/A0-A6,-(A7)
```

```
                  ; Initialize address registers.
                  MOVEA.L    #SIOA_DATA,A0
                  MOVEA.L    #SIOA_CTRL,A1
                  MOVEA.L    #SIOB_DATA,A2
                  MOVEA.L    #SIOB_CTRL,A3
                  MOVEA.L    #BITREV,A4

SIO_BEGIN:  ; Retreive interrupt source information.
                  MOVE.B     #R3,(A1)
                  MOVE.B     (A1),D0

                  ; Check for Channel A Rx interrupt.
                  BTST       #RR3ARIP,D0
                  BEQ.B      SIOIATX

                  ; Get data byte from channel A Rx register.
                  MOVE.B     (A0),D1
                  ANDI.W     #$FF,D1          ; Perform bit reversal.
                  MOVE.B     (A4,D1.W*1),D1

                  ; Check for received XON/XOFF character and set channel A XON/XOFF
                  ; accordingly.
                  CMP.B      #XON,D1
                  BNE.B      1$               ; Not XON character.
                  CLR.B      bATxXoff         ; Set channel A Tx to XON.
                  BRA.B      SIOIATX          ; Continue...

1$:               CMP.B      #XOFF,D1
                  BNE.B      2$               ; Not XOFF character.
                  MOVE.B     #1,bATxXoff      ; Set channel A Tx to XOFF.
                  BRA.B      SIOIATX          ; Continue...
2$:
                  ; Add the character to the receive buffer.
                  MOVEA.L    cpARxBuff,A5     ; Get the pointer to the buffer.
                  MOVE.L     nARxHead,D2      ; Get the head index.
                  MOVE.B     D1,(A5,D2.W*1)   ; Store the data byte.
                  AND.L      #ESCCBUFW,D2     ; Handle wrap around.
                  MOVE.L     D2,nARxHead      ; Save index.

                  ; Check whether we should request XOFF.
                  SUB.L      (nARxTail),D2
                  BGE.B      3$               ; Non-negative result.
                  ADD.L      #ESCCBUFL,D2     ; Negative result, perform mod correction.
3$:               CMP.L      #ESCCXOFT,D2     ; Check threshold.
                  BLT.B      4$               ; No XOFF required.
                  MOVE.B     #XOFFREQ,byAReqXonXoff ; Request XOFF.
4$:
SIOIATX:    ; Check for Channel A Tx interrupt.
                  BTST       #RR3ATIP,D0
                  BEQ.B      SIOIBRX

                  ; Check for Tx XON/XOFF request.
                  CMP.B      #NOXREQ,byAReqXonXoff
                  BEQ.B      1$               ; No XON/XOFF request.
                  CMP.B      #RCVXOFF,byAReqXonXoff
                  BEQ.B      1$               ; No XON/XOFF request.
                  CMP.B      #XOFFREQ,byAReqXonXoff
                  BNE.B      2$
                  MOVE.B     #XOFFBR,(A0)     ; Transmit XOFF
                  BRA.B      4$
2$:               MOVE.B     #XONBR,(A0)      ; Transmit XON
4$:               CLR.B      byAReqXonXoff    ; Clear request flag.
                  BRA.B      SIOIBRX          ; Done processing xmit request.
```

```
                :

1$:
                ; Check XON/XOFF state of port A Tx.
                CMP.B     #0,bATxXoff
                BEQ.B     SIOIBRX        ; Port is XOFF.

                ; Port is XON. Check for new character to xmit.
                MOVE.L    nATxTail,D2    ; Get transmit buffer tail index.
                CMP.L     nATxHead,D2    ; Check against transmit buffer head.
                BEQ.B     SIOIBRX        ; Buffer is empty.
                MOVEA.L   cpATxBuff,A5   ; Get pointer to transmit buffer.
                CLR.L     D3
                MOVE.B    (A5,D2.W*1),D3 ; Get byte.
                MOVE.B    (A4,D3.W*1),(A0) ; Bit reverse and put byte to ESCC.
                ADD.L     #1,D2          ; Increment tail index
                AND.L     #ESCCBUFW,D2   ; Handle wrap around.
                MOVE.L    D2,nATxTail    ; Save index.
SIOIBRX:        ; Check for Channel B Rx interrupt.
                BTST      #RR3BRIP,D0
                BEQ.B     SIOIBTX

                ; Get data byte from channel B Rx register.
                MOVE.B    (A2),D1

SIOIBTX:        ; Check for Channel B Tx interrupt.
                BTST      #RR3BTIP,D0
                BEQ.B     SIOIDONE
                NOP

SIOIDONE:       ; Restore registers.
                MOVEM.L   (A7)+,D0-D7/A0-A6

                RTE

                SECTION   3     ; Section 3 is unitialized static data storage.
cARxBuffer:     DS.B ESCCBUFL ; ESCC Port A receive buffer.
cATxBuffer:     DS.B ESCCBUFL ; ESCC Port B transmit buffer.
cBRxBuffer:     DS.B ESCCBUFL ; ESCC Port A receive buffer.
cBTxBuffer:     DS.B ESCCBUFL ; ESCC Port B transmit buffer.

cpARxBuff:      DS.L 1        ; Pointer to port A Rx buffer.
nARxHead:       DS.L 1        ; Port A head index.
nARxTail:       DS.L 1        ; Port A tail index.
cpATxBuff:      DS.L 1        ; Pointer to port A Rx buffer.
nATxHead:       DS.L 1        ; Port A head index.
nATxTail:       DS.L 1        ; Port A tail index.
bATxXoff:       DS.B 1        ; Port A transmit disable boolean flag.
byAReqXonXoff:  DS.B 1        ; Port A transmit XON/XOFF request flag.

cpBRxBuff:      DS.L 1        ; Pointer to port A Rx buffer.
nBRxHead:       DS.L 1        ; Port A head index.
nBRxTail:       DS.L 1        ; Port A tail index.
cpBTxBuff:      DS.L 1        ; Pointer to port A Rx buffer.
nBTxHead:       DS.L 1        ; Port A head index.
nBTxTail:       DS.L 1        ; Port A tail index.
bBTxXoff:       DS.B 1        ; Port A transmit disable boolean flag.
byBReqXonXoff:  DS.B 1        ; Port A transmit XON/XOFF request flag.

                SECTION   0 ; Section 0 is the code section.
;----------------------------------------------------------------
; SCSI INTERRUPT HANDLER
;
; Description:
; This interrupt services the SCSI port.
SCSI_INTERRUPT:
```

```
        ; Save registers.
        MOVEM.L   D0-D7/A0-A6,-(A7)

        ; Pre-load address registers.
        MOVEA.L   #BITREV,A0
        MOVEA.L   #SCSI_PORT,A1

        ; Read SCSI Status Register to determine cause of interrupt.
        CLR.L     D0
        MOVE.B    (SCS_STATS,A1),D0   ; Get status
        MOVE.B    (A0,D0.W*1),D0      ; Perform bit reversal.
        MOVE.B    D0,bySCSIStatus     ; Shadow status to status location.
        MOVE.B    D0,D1               ; Save status value.

        ; Check for reset interrupt.
        AND.B     #$F0,D0             ; Check status field of SCSI status.
        BEQ.B     SCSIDONE            ; If zero then we're done.

        ; Check for a command completed successfully interrupt.
        AND.B     #$10,D0             ; Check status field of SCSI status.
        BNE.B     SCSIPAUSI
        BRA.B     SCSIDONE            ; This shouldn't occur so we're done.

SCSIPAUSI:
        ; Check for a command pause/abort interrupt.
        MOVE.B    D1,D0               ; Get status.
        AND.B     #$20,D0             ; Check status field of SCSI status.
        BNE.B     SCSIABEND

        BRA.B     SCSIDONE            ; That's all folks.

SCSIABEND:
        ; Check for a premature termination interrupt.
        MOVE.B    D1,D0               ; Get status.
        AND.B     #$40,D0
        BNE.B     SCSISERVICE

        BRA.B     SCSIDONE            ; That's all folks.

SCSISERVICE:
        ; An event on the SCSI bus requires service.
        ; The possibilities are CODE in {$2,$3,$4,$7}.

        MOVE.B    D1,D0               ; Get status.
        AND.B     #$F,D0              ; Mask status, leave code.
        CMP.B     #$2,D0              ; Check for sel w/o ATN.
        BNE.B     SCSSERVSELA         ; No.
                                      ; Yes.

        ; Get data of interest.
        CLR.L     D0
        MOVE.B    (SCS_SRCID,A1),D0   ; Read Source ID Register.
        MOVE.B    (A0,D0.W*1),D0      ; Perform bit reversal.
        MOVE.B    D0,D1               ; Copy contents of SRCID register.
        AND.B     #$7,D0              ; Mask out SRCID value.
        MOVE.B    D0,bySCSSourceID    ; Save SRCID value.
        AND.B     #$8,D0              ; Mask out SRCID valid bit.
        BNE.B     1$
        CLR.B     bSCSSIDValid        ; Write false to valid flag.

        BRA.B     2$
1$:     MOVE.B    #$1,bSCSSIDValid    ; Write true to valid flag.
2$:

        ; Issue XFER COUNT Register and Issue RECEIVE COMMAND command.
        ; Prepare to receive a six byte command.
        MOVE.B    #$0,(SCS_TRCR2,A1)  ; Write zero to MSB.
        MOVE.B    #$0,(SCS_TRCR1,A1)  ; Write zero to 2nd MSB.
        MOVE.B    #$6,(SCS_TRCR0,A1)  ; Write 6 to LSB.
```

```
            MOVE.B      #SBIC_RCCMD,(SCS_CMDRG,A1) ; Issue RCV CMD command.

            BRA.B       SCSIDONE            ; That's all folks.
SCSSERVSELA:
            CMP.B       #$3,D0              ; Check for sel w/ ATN.
            BNE.B       SCSSERVATN          ; No.
          .; Yes, but this isn't supported.
            ; This only happens when an identify message is being transmitted.
            BRA.B       SCSIDONE            ; That's all folks.
SCSSERVATN:
            CMP.B       #$4,D0              ; Check for ATN assertion.
            BNE.B       SCSSERVUNK          ; No.
                                           ; Yes.
            BRA.B       SCSIDONE            ; That's all folks.
SCSSERVUNK:                                ; Must be WFSR paused.
            NOP
SCSIDONE: ; Restore registers.
            MOVEM.L     (A7)+,D0-D7/A0-A6
            RTE
            SECTION   3       ; Section 3 is unitialized static data storage.
bySCSSourceID: DS.B 1         ; SCSI Source ID value.
bSCSSIDValid:  DS.B 1         ; SCSI Source ID valid flag.
SCSCmdFrame:   DS.B 12        ; SCSI Command frame storage.
bySCSIStatus:  DS.B 1         ; SCSI Status register value.
bySCSIPhase    DS.B 1         ; SCSI Phase state variable.
            SECTION   0       ; Section 0 is the code section.
;-------------------------------------------------------------------
; BUS ERROR INTERRUPT HANDLER
;
; Description:
; This interrupt services the bus error exception.
BERR_INTERRUPT:
            ADDQ.L      #1,(dwBusErrorCount) ; Inc. bus error counter.
            RTE
            SECTION   3       ; Section 3 is unitialized static data storage.
dwBusErrorCount:  DS.L 1  ; Bus error counter.
            SECTION   0 ; Section 0 is the code section.
;-------------------------------------------------------------------
; IPL LEVEL SET ROUTINE
; void SetIPL(int iIPL)
;
; Description:
; This routine sets the processor IPL level. This routine uses the
; C language calling structure.
; The prototype for this function is void SetIPL(int);
SetIPL    PROC
            ; Save data registers.
            MOVEM.L     D0-D1,-(A7)

            ; Get status register.
            MOVE        SR,D0

            ; Mask out IPL bits.
            AND.W       #$F8FF,D0
```

```
        ; Get IPL level.
        MOVE.L    (12,A7),D1
        AND.W     #$700,D1

        ; Mask IPL level into SR.
        EOR.W     D1,D0

        ; Put SR back.
        MOVE      D0,SR

        ; Restore data registers.
        MOVEM.L   (A7)+,D0-D1

        ; Return from the procedure.
        RTS


;   ----------
;   CONSTANTS
;   ----------

        END
```

D.10   POST and Initialization: POSTINIT.M68

```
; Description: POST for Backplane Firmware
;   -------------
; Major Rev: 0
; Minor Rev: 0
; Date: 5/29/92
;
; Author: Jon Mellott
;
; High Speed Digital Architecture Laboratory
; University of Florida
; 405 CSE Building
; Gainesville, FL  32611

;   --------------  --------
;   IDENTIFICATION DIVISION
;   --------------  --------
POSTINIT IDNT 0,1          ; Module POSTINIT, Major rev=0, Minor rev=0
        SECTION 0          ; Place in section zero (ROM)

;   -----------  --------
;   ENVIRONMENT DIVISION
;   -----------  --------

        OPT       NOMEX,P=68030

;   --------  ----------
;   EXTERNAL REFERENCES
;   --------  ----------

        XREF .CENTRY       ; Startup module reference.
        XREF dwBusErrorCount ; Bus error counter.

;   -------
;   EXPORTS
;   -------

        XDEF BITREV        ; Byte wide bit-reversal table.
        XDEF POSTINIT      ; POST entry point.

;   -------
;   EQUATES
;   -------

; Base location of RAM.
```

```
RAM_BASE  EQU       $100000

; InvestiGATOR I/O port location and misc. constants.
          INCLUDE   INVESTIO.INC

; ----- -----------
; MACRO DEFINITIONS
; ----- -----------
;------------------------------------------------------------------
; STRING PRINT MACRO
;
; Description:
; The following macro takes as its argument the address of a text string
; and causes the text string to be printed to serial port A.
; This macro uses no transient storage and does not save registers used.
; REGISTER USAGE: A3,A4,A5,A6,D7
HPRINTM   MACRO
          MOVEA.L   #\1,A3        ; Load the base address of the message.
          MOVEA.L   #SIOA_CTRL,A4 ; Load the SIO control port A addr -> A4.
          MOVEA.L   #SIOA_DATA,A5 ; Load the SIO data port A addr -> A5.
          MOVEA.L   #BITREV,A6    ; Load the location of the bit rev. table.
HPRM1\@:  MOVE.B    #R0,(A4)      ; Read status word and check for Tx buffer
          MOVE.B    (A4),D7       ; empty status.
          ANDI.B    #$20,D7       ; Mask out Tx buffer empty bit.
          BEQ.B     .HPRM1\@      ; If not empty then wait.

          CLR.L     D7            ; Initialize register.
          MOVE.B    (A3)+,D7      ; Read byte.
          CMP.B     #0,D7         ; Check for null character.
          BEQ.B     HPRM2\@       ; Null character.
          MOVE.B    (A6,D7),(A5)  ; Non-null, so write to SCC.
          BRA       HPRM1\@       ; Loop back...
HPRM2\@:
          ENDM

;------------------------------------------------------------------
; ADDRESS REGISTER PRINT MACRO
;
; Description:
; The following macro takes as its argument an address register and
; prints the contents of that address register.
; This macro uses no transient storage and does not save registers used.
; REGISTER USAGE: A6,A5,A4,A3,D7,D6,D5
ARPRNM    MACRO
          MOVE.L    \1,D7         ; Save address register in question.
          MOVEA.L   #HEXTABLE,A3  ; Load the ASCII hex table addr -> A3.
          MOVEA.L   #SIOA_CTRL,A4 ; Load the SIO control port A addr -> A4.
          MOVEA.L   #SIOA_DATA,A5 ; Load the SIO data port A addr -> A5.
          MOVEA.L   #BITREV,A6    ; Load the location of the bit rev. table.
          MOVE.B    #8,D5         ; Number of bytes to print.
ARPRNM\@: MOVE.B    #R0,(A4)      ; Read status word and check for Tx buffer
          MOVE.B    (A4),D6       ; empty status.
          ANDI.B    #$20,D6       ; Mask out Tx buffer empty bit.
          BEQ.B     ARPRNM\@      ; If not empty then wait.

          ROL.L     #4,D7         ; Rotate MS nibble to LS nibble position.
          MOVE.L    D7,D6         ; Copy rotated data.
          ANDI.L    #$F,D6        ; Mask data down to hex table offset.
          MOVE.B    (A3,D6),D6    ; Get ASCII value of value.
          MOVE.B    (A6,D6),(A5)  ; Write SCC.
```

```
            SUBQ.B    #1,D5            ; Decrement loop counter.
            BNE.B     ARPRNM\@         ; Loop back.

            ENDM

;-------------------------------------------------------------------
;  DATA PRINT MACRO
;
;  Description:
;  The following macro takes as its argument a valid longword argument and
;  prints the value.
;  This macro uses no transient storage and does not save registers used.
;  REGISTER USAGE: A6,A5,A4,A3,D7,D6,D5
DPRNM       MACRO
            MOVE.L    \1,D7            ; Save address register in question.
            MOVEA.L   #HEXTABLE,A3     ; Load the ASCII hex table addr -> A3.
            MOVEA.L   #SIOA_CTRL,A4    ; Load the SIO control port A addr -> A4.
            MOVEA.L   #SIOA_DATA,A5    ; Load the SIO data port A addr -> A5.
            MOVEA.L   #BITREV,A6       ; Load the location of the bit rev. table.
            MOVE.B    #8,D5            ; Number of bytes to print.

DPRNM\@:    MOVE.B    #R0,(A4)         ; Read status word and check for Tx buffer
            MOVE.B    (A4),D6          ; empty status.
            ANDI.B    #$20,D6          ; Mask out Tx buffer empty bit.
            BEQ.B     DPRNM\@          ; If not empty then wait.

            ROL.L     #4,D7            ; Rotate MS nibble to LS nibble position.
            MOVE.L    D7,D6            ; Copy rotated data.
            ANDI.L    #$F,D6           ; Mask data down to hex table offset.
            MOVE.B    (A3,D6),D6       ; Get ASCII value of value.
            MOVE.B    (A6,D6),(A5)     ; Write SCC.

            SUBQ.B    #1,D5            ; Decrement loop counter.
            BNE.B     DPRNM\@          ; Loop back.

            ENDM

;-------------------------------------------------------------------
;  MARCH-UP READ MACRO
;
;  Description:
;  The following macro performs a march up read of RAM. It takes as its
;  argument the value which is expected to be found in memory.
;  This macro uses no transient storage and does not save registers used.
;  REGISTER USAGE: A0,D2,D1,D0
MUPRM       MACRO
            MOVE.L    \1,D0            ; Expected value.
            MOVEA.L   #RAM_BASE,A0     ; Load memory base address to A0.
            MOVE.L    #$40000,D1       ; Load loop counter.
MUPRM\@:    MOVE.L    (A0),D2          ; Get the data at (A0).
            CMP.L     D2,D0            ; Check data.
            BEQ.W     MUPRM1\@         ; Data is o.k.
            HPRINTM   STR4             ; Not ok, print fault message.
            ARPRNM    A0
            HPRINTH   STR5
            DPRNM     D2
            HPRINTH   STR6
MUPRM1\@:   ADDQ.L    #4,A0            ; Increment address counter.
            SUBQ.L    #1,D1            ; Decrement countdown counter.
            BNE.W     MUPRM\@

            ENDM

;-------------------------------------------------------------------
```

```
; MARCH-UP WRITE MACRO
;
; Description:
; The following macro performs a march up write of RAM. It takes as its
; argument the value which is to be written to memory.
; This macro uses no transient storage and does not save registers used.
; REGISTER USAGE: A0,D1,D0
MUPWM       MACRO
            MOVE.L    \1,D0           ; Value to write.
            MOVEA.L   #RAM_BASE,A0    ; Load memory base address to A0.
            MOVE.L    #$40000,D1      ; Load loop counter.
MUPWM\@:    MOVE.L    D0,(A0)+        ; Write data to (A0).
            SUBQ.L    #1,D1           ; Decrement countdown counter.
            BNE.B     MUPWM\@
            ENDM
;-------------------------------------------------------------------
; MARCH-UP READ,WRITE MACRO
;
; Description:
; The following macro performs a march up read then write of RAM. It takes
; as its arguments the value which expected to be in memory and the
; value which is to be written to memory.
; This macro uses no transient storage and does not save registers used.
; REGISTER USAGE: A0,D0,D1,D2,D3
MUPRWM      MACRO
            MOVE.L    \1,D0           ; Expected value.
            MOVE.L    \2,D1           ; New value.
            MOVE.L    #$40000,D3      ; Count-down counter.
            MOVEA.L   #RAM_BASE,A0    ; Load memory base address to A0.
MUPRWM\@:   MOVE.L    (A0),D2         ; Get the data at (A0).
            CMP.L     D2,D0           ; Compare the data with the expected value.
            BEQ.W     MUPRWM1\@       ; OK
            HPRINTM   STR4            ; Not ok, print fault message.
            ARPRNM    A0
            HPRINTM   STR5
            DPRNM     D2
            HPRINTH   STR6
MUPRWM1\@:
            MOVE.L    D1,(A0)+        ; Place new value in memory.
            SUBQ.L    #1,D3           ; Decrement countdown counter.
            BNE.W     MUPRWM\@        ; Loop-back.
            ENDM
;-------------------------------------------------------------------
; MARCH-DOWN READ MACRO
;
; Description:
; The following macro performs a march down read of RAM. It takes as its
; argument the expected value which is to be read from memory.
; This macro uses no transient storage and does not save registers used.
; REGISTER USAGE: A0,D2,D1,D0
MDRM        MACRO
            MOVE.L    \1,D0           ; Place expected value in D0.
            MOVE.L    #$3FFFF,D1      ; Load initial longword index into D1.
            MOVEA.L   #RAM_BASE,A0    ; Load memory base address to A0.
MDRM\@:     MOVE.L    (A0,D1.L*4),D2  ; Get value from memory.
            CMP.L     D2,D0           ; Compare the data with the expected value.
            BEQ.W     MDRM1\@         ; OK.
            HPRINTM   STR3            ; Not ok, print fault message.
```

```
                DPRNM    D1
                HPRINTH  STR5
                DPRNM    D2
                HPRINTM  STR6
      MDRM1\@:  SUBQ.L   #1,D1            ; Decrement index/loop counter.
                BGE.W    MDRM\@           ; Loop back.

                ENDM
;------------------------------------------------------------------
; MARCH-DOWN READ,WRITE MACRO
;
; Description:
; The following macro performs a march down read then write of RAM. It takes
; as its arguments the value which is expected and the new value.
; This macro uses no transient storage and does not save registers used.
; REGISTER USAGE: A0,D3,D2,D1,D0

      MDRWM     MACRO
                MOVE.L   \1,D0            ; Place expected value in D0.
                MOVE.L   \2,D3            ; Place new value in D3.
                MOVE.L   #$3FFFF,D1       ; Load initial longword index into D1.
                MOVEA.L  #RAM_BASE,A0     ; Load memory base address to A0.
      MDRWM\@:  MOVE.L   (A0,D1.L*4),D2   ; Get value from memory.
                CMP.L    D2,D0            ; Compare the value with expected.
                BEQ.W    MDRWM1\@         ; OK.
                HPRINTM  STR3             ; Not ok, print fault message.
                DPRNM    D1
                HPRINTH  STR5
                DPRNM    D2
                HPRINTM  STR6
      MDRWM1\@: MOVE.L   D3,(A0,D1.L*4)   ; Write new value.
                SUBQ.L   #1,D1            ; Decrement index/loop counter.
                BGE.W    MDRWM\@          ; Loop back.

                ENDM
; ---- -------
; MAIN PROGRAM
; ---- -------
;
; The following code segment performs these initialization tasks:
;
; 1. Initialize ESCC for POST messages.
; 2. Initialize and test DRAM.
;    a. Perform DRAM charge pump initialization
;    b. Perform DRAM tests.
; 3. Enable the cache memory
; 4. Initialize ESCC driver and interrupt handler.
; 5. Initialize SBIC driver and interrupt handler.
; 6. Initialize misc. interrupt service routines.
; 7. Enable interrupts
; 8. Execute startup code.
;
                SECTION  0                ; Section 0 is the code segment.

      POSTINIT:
;
; Initialize serial port.
;
      SIOINIT:  MOVEA.L  #SIOA_DATA,A0    ; Load SIOA data port address into A0.
                MOVEA.L  #SIOA_CTRL,A1    ; Load SIOA control port address into A1.

                MOVE.B   #R9,(A1)         ; Write $C0 to WR9 to perform hardware
                MOVE.B   #$3,(A1)         ; reset.
```

```
        MOVE.B    #R4,(A1)      ; Write $44 to WR4 to enable 16x clock,
        MOVE.B    #$22,(A1)     ; one stop bit, and no parity.

        MOVE.B    #R3,(A1)      ; Write $C0 to set Rx 8 data bits,
        MOVE.B    #$3,(A1)      ; Rx disabled.

        MOVE.B    #R5,(A1)      ; Write $60 to set Tx 8 data bits,
        MOVE.B    #$6,(A1)      ; DTR,RTS,Tx off.

        MOVE.B    #R9,(A1)      ; Write $0 to WR9 to disable interrupts.
        MOVE.B    #$0,(A1)

        MOVE.B    #R10,(A1)     ; Write $0 to WR10 for NRZ coding.
        MOVE.B    #$0,(A1)

        MOVE.B    #R11,(A1)     ; Write $56 so Tx & Rx = BRG output,
        MOVE.B    #$6A,(A1)     ; TRxC = BRG output.

        MOVE.B    #R12,(A1)     ; Set baud rate LSB = $43 for 4800 baud.
        MOVE.B    #$C2,(A1)
        MOVE.B    #R13,(A1)     ; Set baud rate MSB = $0 for 4800 baud.
        MOVE.B    #$00,(A1)

        MOVE.B    #R14,(A1)     ; Write $12 to WR14 to set BRG in = /RTxC,
        MOVE.B    #$8,(A1)      ; set BRG clk=PCLK, and en. local loopback.

        MOVE.B    #R14,(A1)     ; Write $13 to WR14 to enable baud rate
        MOVE.B    #$C8,(A1)     ; generator.

        MOVE.B    #R3,(A1)      ; Write $C1 to WR3 to enable receiver.
        MOVE.B    #$83,(A1)

        MOVE.B    #R5,(A1)      ; Write $EA to WR5 to enable transmitter,
        MOVE.B    #$57,(A1)     ; set DTR and CTS.
        ; INITIALIZATION OF PORT A DONE.

;
; Print heading message...
;
        HPRINTM   STR0

;
; Initialize the dynamic RAM.
;
SCRAMINIT:
;
; Print DRAM initializing message...
;
        HPRINTM   STR1
        MOVE.B    #8,D0         ; Number of times to read all locations.
DRAM_INIT1:
        MOVEA.L   #RAM_BASE,A0  ; Init RAM address base register.
        MOVE.L    #$40000,D1    ; Init count-down counter.
DRAM_INIT2:
        MOVE.L    (A0)+,D2      ; Read the mem at location pointed at by A0.
        SUBQ.L    #1,D1         ; Decrement the counter.
        BNE       DRAM_INIT2    ; Loop back for next location.

        SUBQ.B    #1,D0         ; Decrement pump count.
        BNE       DRAM_INIT1    ; Loop back if not done.

;
; SCRAM Test.
;
SCRAMTST:

; Fourth SCRAM Test
; This test writes unique longword values to the SCRAM in ascending order
```

```
        :

; from zero to $3FFFF.
            HPRINTM   STR23

            ; WRITE DATA
            CLR.L     D0            ; Initialize test sequence counter.
            MOVEA.L   #RAM_BASE,A0  ; Initialize RAM address base register.
            MOVE.L    #$40000,D1    ; Initialize countdown counter.
RAMT4DW:    MOVE.L    D0,(A0)+      ; Write the data.
            ADDQ.L    #1,D0         ; Increment the data register.
            SUBQ.L    #1,D1         ; Decrement the countdown register.
            BNE.B     RAMT4DW       ; Loop back.

            ; READ DATA
            CLR.L     D0            ; Initialize test sequence counter.
            MOVEA.L   #RAM_BASE,A0  ; Initialize RAM address base register.
            MOVE.L    #$40000,D1    ; Initialize countdown counter.
            CLR.L     D2            ; Clear fault counter.
RAMT4DR:    CMP.L     (A0),D0       ; Check the data.
            BEQ.W     RAMT4DX       ; O.K.
            ADDQ.L    #1,D2         ; Increment fault counter.
            HPRINTM   STR4          ; Print fault message.
            ARPRNM    A0
            HPRINTM   STR5
            DPRNM     (A0)
            HPRINTM   STR6

RAMT4DX:    ADDQ.L    #4,A0         ; Increment address counter.
            ADDQ.L    #1,D0         ; Increment test sequence counter.
            SUBQ.L    #1,D1         ; Decrement the countdown register.
            BNE.W     RAMT4DR       ; Loop back.

            ; PRINT REPORT
            CMP.L     #0,D2         ; Check fault counter.
            BNE.B     RAMT4F        ; Failed.

            HPRINTM   STR12         ; Passed.
            BRA.B     RAMT4E
RAMT4F:     HPRINTM   STR24         ; Print failure message.
RAMT4E:

; Sixth SCRAM Test
; This test looks for a stuck-at fault on the address multiplexer lines.
; This test is accomplished by the following algorithm:
;
;    for i:=2 to 19
;      Memory[0]:=0;
;      Memory[2^i]:=1;
;      if Memory[0]!=0 then there exists a fault.
;    end;
RAMT6:      HPRINTM   STR26

            MOVE.L    #0,D1         ; Load zero into D1.
            MOVE.L    #$FFFFFFFF,D2 ; Load all ones into D2.
            MOVE.L    #1,D3         ; Initial 4*2^i value.
            MOVE.L    #19,D0        ; Number of times to loop.
            MOVEA.L   #RAM_BASE,A0  ; Load RAM base address into A0.
RAMT6L:     MOVE.L    D1,(A0)       ; Memory[0]:=0.
            MOVE.L    D2,(A0,D3.L*4) ; Memory[2^i]:=1.
            LEA.L     (A0,D3.L*4),A1 ; Loading effective address into A1.
            HPRINTM   STR37
            ARPRNM    A1            ; Print effective address value.
            HPRINTM   STR38
            CMP.L     (A0),D1       ; Check for fault.
            BEQ.B     RAMT6OK       ; Passed.
```

```
            HPRINTM  STR27          ; Failed.
            BRA.B    RAMT6C         ; Continue.
RAMT6OK:    HPRINTM  STR28          ; Passed.
RAMT6C:     LSL.L    #1,D3          ; Shift address offset. (++i).
            ANDI.L   #$3FFFF,D3     ; Make sure this offset is not fubar.
            SUBQ.L   #1,D0          ; Decrement loop counter.
            BNE.W    RAMT6L         ; Loop back.
; Seventh SCRAM Test
; This test performs the March C test described in ACM Computing Surveys,
; vol. 22, no. 1.
;
; The March C test is described by the following sequence of march elements:
;          ^(r,w1); ^(r,w0); ^(r); v(r,w1); v(r,w0); v(r)
; The memory is initialized by a ^(w0) march.

RAMT7:      HPRINTM  STR29

            MUPWM    #0             ; Initialize by ^(w0).

            HPRINTM  STR30          ; Print M1 string...
            MUPRWM   #0,#$FFFFFFFF  ; Perform ^(r0,w1).

            HPRINTM  STR31          ; Print M2 string...
            MUPRWM   #$FFFFFFFF,#0  ; Perform ^(r1,w0).

            HPRINTM  STR32          ; Print M3 string...
            MUPRM    #0             ; Perform ^(r0).

            HPRINTM  STR33          ; Print M4 string...
            MDRWM    #0,#$FFFFFFFF  ; Perform v(r0,w1)

            HPRINTM  STR34          ; Print M5 string...
            MDRWM    #$FFFFFFFF,#0  ; Perform v(r1,w0);

            HPRINTM  STR35          ; Print M6 string...
            MDRM     #0             ; Perform v(r0).

;
; Infinite loop.
;
            BRA      SCRAMTST       ; Loop-back for safety.

;
; Initialize miscellaneous ISRs.
;
            ; Initialize bus error exception counter.
            CLR.L    dwBusErrorCount

;
; Enable cache memory.
;

;
; Enable interrupts.
;

;
; Goto startup code.
;
            BRA.W    .CENTRY        ; Goto C startup code entry point.

; ---------
; CONSTANTS
; ---------
            SECTION  1              ; Section 1 is the text segment.
;
; String constants.
;
STR0:       DC.B     'InvestiGATOR POST/Init version 0.2',10,13
            DC.B     'Jon Hellott, 5/27/92W',10,13,10,13
```

```
            DC.B     'High Speed Digital Architecture Laboratory',10,13
            DC.B     'University of Florida',10,13
            DC.B     'Gainesville, FL 32611',10,13,10,13
            DC.B     'InvestiGATOR Initializing and Testing...',10,13,0
STR1:       DC.B     'SCRAM Initializing...',10,13,0

STR3:       DC.B     '  ***FAULT: LI(',0
STR4:       DC.B     '  ***FAULT: @(',0
STR5:       DC.B     ')=',0
STR6:       DC.B     '.',10,13,0

STR12:      DC.B     ' Test passed.'
            DC.B     10,13,0
STR23:      DC.B     'SCRAM Test #4...',10,13,0
STR24:      DC.B     '  ***FAULT: SCRAM Test #4 Failed.',10,13,0
STR25:      DC.B     'SCRAM Test #5...',10,13,0
STR26:      DC.B     'SCRAM Test #6: Address MUX SAF Test...',10,13,0
STR27:      DC.B     '***FAULT: SAF.',10,13,0
STR28:      DC.B     'Passed.',10,13,0
STR37:      DC.B     '  (',0
STR38:      DC.B     '): ',0
STR29:      DC.B     'SCRAM Test #7: March C Test...',10,13,0
STR30:      DC.B     '  M1...',10,13,0
STR31:      DC.B     '  M2...',10,13,0
STR32:      DC.B     '  M3...',10,13,0
STR33:      DC.B     '  M4...',10,13,0
STR34:      DC.B     '  M5...',10,13,0
STR35:      DC.B     '  M6...',10,13,0
STR36:      DC.B     '***FAULT',10,13,0

; ASCII HEXADECIMAL TABLE
HEXTABLE: DC.B       '0123456789ABCDEF'

; BIT REVERSAL TABLE.
BITREV:   DC.B       $00,$80,$40,$C0,$20,$A0,$60,$E0
          DC.B       $10,$90,$50,$D0,$30,$B0,$70,$F0
          DC.B       $08,$88,$48,$C8,$28,$A8,$68,$E8
          DC.B       $18,$98,$58,$D8,$38,$B8,$78,$F8
          DC.B       $04,$84,$44,$C4,$24,$A4,$64,$E4
          DC.B       $14,$94,$54,$D4,$34,$B4,$74,$F4
          DC.B       $0C,$8C,$4C,$CC,$2C,$AC,$6C,$EC
          DC.B       $1C,$9C,$5C,$DC,$3C,$BC,$7C,$FC
          DC.B       $02,$82,$42,$C2,$22,$A2,$62,$E2
          DC.B       $12,$92,$52,$D2,$32,$B2,$72,$F2
          DC.B       $0A,$8A,$4A,$CA,$2A,$AA,$6A,$EA
          DC.B       $1A,$9A,$5A,$DA,$3A,$BA,$7A,$FA
          DC.B       $06,$86,$46,$C6,$26,$A6,$66,$E6
          DC.B       $16,$96,$56,$D6,$36,$B6,$76,$F6
          DC.B       $0E,$8E,$4E,$CE,$2E,$AE,$6E,$EE
          DC.B       $1E,$9E,$5E,$DE,$3E,$BE,$7E,$FE
          DC.B       $01,$81,$41,$C1,$21,$A1,$61,$E1
          DC.B       $11,$91,$51,$D1,$31,$B1,$71,$F1
          DC.B       $09,$89,$49,$C9,$29,$A9,$69,$E9
          DC.B       $19,$99,$59,$D9,$39,$B9,$79,$F9
          DC.B       $05,$85,$45,$C5,$25,$A5,$65,$E5
          DC.B       $15,$95,$55,$D5,$35,$B5,$75,$F5
          DC.B       $0D,$8D,$4D,$CD,$2D,$AD,$6D,$ED
          DC.B       $1D,$9D,$5D,$DD,$3D,$BD,$7D,$FD
          DC.B       $03,$83,$43,$C3,$23,$A3,$63,$E3
          DC.B       $13,$93,$53,$D3,$33,$B3,$73,$F3
          DC.B       $0B,$8B,$4B,$CB,$2B,$AB,$6B,$EB
          DC.B       $1B,$9B,$5B,$DB,$3B,$BB,$7B,$FB
          DC.B       $07,$87,$47,$C7,$27,$A7,$67,$E7
```

238

```
        DC.B    $17,$97,$57,$D7,$37,$B7,$77,$F7
        DC.B    $0F,$8F,$4F,$CF,$2F,$AF,$6F,$EF
        DC.B    $1F,$9F,$5F,$DF,$3F,$BF,$7F,$FF

        END
```

D.11 C Startup Code: STARTUP.M68

```
*********************************************************************
*                                                                   *
*                       Copyright (C) 1987 by                       *
*                  Production Languages Corporation                 *
*                                                                   *
*                          P.O. Box 109                             *
*                  Weatherford, Texas 76086-0109                    *
*                         (817) 599-8363                            *
*                                                                   *
*********************************************************************
STARTUP    IDNT 0,0
  SECTION  0

  INCLUDE        ..\config\config

*
*          Generic C Program Start-up code
*

  XDEF .CENTRY
  XDEF .maxfiles
  XDEF .file
  XDEF .s_errno
  XDEF errno
  XDEF .stksize
  XDEF .hpsize
  XDEF .argc
  XDEF .argv
  XDEF .tmpstk
  XDEF .endcode
  XDEF .next_rand

  XREF .stack
  XREF STACKSZ
  XREF .hptop
  XREF .hpbot
  XREF .allocp
  XREF exit
  XREF main
  XREF abort
  XREF .init

  IFNE CMDLINE
  XREF .u_scancmd
  ENDC

  IFNE SIGNAL
  XDEF .sigvect
  ENDC

  XREF .s_siginit

  IFNE FILEIO
  XREF .s_fsinit
  ENDC

*       This label must be the lowest address in the
```

239

```
*          code/data segment for the automatic updating
*          of position independant initializers to work.
.CENTRY:
  IFNE  POSIND & (M68000 | M68010)
*
*          Get the base address.
*          This code assumes that .CENTRY is the
*          lowest address of the program.
*          The register used here MUST match the
*          base register used in CONFIG.M68.
*
LEA.L     .CENTRY,A5          ; Get base address
  ENDC

; The following section added by J. Mellott, 2/9/92SUN.
; This performs dynamic RAM initialization. The TC514258
; Static Column RAM requires eight refresh cycles before
; the memory may be used reliably. These refresh cycles may
; be forced by performing reads. This block of code ensures
; that the memory is initialized before it is used.
;
; Note that the following code segment has the amount of SCRAM
; and its location hard-wired in place.
;
  MOVE.B   #8,D0          ; Number of times to read all locations.
DRAM_INIT1:
  MOVEA.L  #$100000,A0  ; Init RAM address base register.
  MOVE.L   #$40000,D1   ; Init count-down counter.
DRAM_INIT2:
  MOVE.L   (A0)+,D2     ; Read the mem at location pointed at by A0.
  SUB.L    #1,D1        ; Decrement the counter.
  BNE      DRAM_INIT2   ; Loop back for next location.

  SUB.L    #1,D0        ; Decrement pump count.
  BNE      DRAM_INIT1   ; Loop back if not done.
;
; SCRAM initialization finished.
;
GETADDR  ENDREGS,A7        ; Get end address of .INIREGS
MOVEM.L  D0-D7/A0-A6,-(A7) ; Save away register contents

  IFNE  M68881
FMOVE.L  #$FC00,FPCR        ; Initialize 68881
  ENDC

*          Clear out bss (unitialized) data area
GETADDR  .bssbeg,A0        ; Get address beginning
GETADDR  .bssend,A1        ; Get address of end
clrbss:
CMPA.L   A0,A1            ; Done?
BEQ.B    bssdone         ; Yes, jump
CLR.B    (A0)+           ; Clear memory
BRA.B    clrbss          ; Go again
bssdone:
*          Update position independant initializers
GETADDR  .CENTRY,A1       ; Get address of segment
CMPA.L   #0,A1           ; Is program loaded at 0?
BEQ.B    initdone        ; Yes, skip initialization
GETADDR  .beginit,A0      ; start of initializer list
ADDQ     #4,A0           ; Skip DC.L 0
GETADDR  .endcode,A3      ; Get address of end
doinit:
```

```
        CMPA.L    A3,A0           ; Done?
        BEQ.B     initdone        ; Yes, jump
        MOVEA.L   (A0)+,A2        ; Get list entry
        ADDA.L    A1,A2           ; Add in base for pointer
        MOVE.L    (A2),D0         ; Get address to update
        ADD.L     A1,D0           ; Add in base
        MOVE.L    D0,(A2)         ; Put back address
        BRA.B     doinit          ; Go again
initdone:
*           Initialize RTL modifiable variables
* The MAXFILE and STACKSZ references in the following code were changed
* to immediate operands 2/16/92 --JDM
        GETADDR  .maxfiles,A0           * Maximum open files
        MOVE.L   #MAXFILE,(A0)          *
        GETADDR  .hpsize,A0             * Size of heap segments
        MOVE.L   #STACKSZ,(A0)          *
        GETADDR  .stksize,A0            * Size of stack segment
        MOVE.L   #STACKSZ,(A0)          *
        GETADDR  .next_rand,A0          * Random number seed
        MOVE.L   #1,(A0)                *
*           Set up temporary stack and call _u_scancmd()
        GETADDR  .tmpstk,A7             ; Set up temporary stack
    IFNE  CMDLINE
 GETADDR  .argv,A0
        MOVE.L   A0,-(SP)        ; Pass address of _argv
        GETADDR  .argc,A0
        MOVE.L   A0,-(SP)        ; Pass address of _argc
        GETADDR  .cmdline,A0
        MOVE.L   A0,-(SP)        ; Pass address of _cmdline
        CALL     .u_scancmd,A2   ; Call scancmnd()
        ADDA.L   #12,SP
    ENDC

*           Call _init() so that it can update _stksize and
*           _maxfiles if desired
        GETADDR  .argv,A0
        MOVE.L   A0,-(SP)        ; Pass address of _argv
        GETMEM   .argc,D0,L,A1
        MOVE.L   D0,-(SP)        ; Pass _argc
        CALL     .init,A2          ; call _init
        ADDQ.L   #8,SP
*           Set up real stack
        GETADDR  .stack,A7       ; Address of stack segment
        GETADDR  .stkaddr,A0
        MOVE.L   A7,(A0)         ; Store top of stack
*           physical address of segment is in A0, length is in D0
        GETADDR  .allocp,A0
        CLR.L    (A0)            ; Clear heap list pointer
        GETADDR  .hpbot,A0
        MOVE.L   A7,(A0)         ; Set bottom of heap
        GETADDR  .hptop,A0
        MOVE.L   A7,(A0)         ; Set top of heap
        ADDA.L   #STACKSZ,A7     ; Calc hi address of stack seg
*           Initialize file system
    IFNE  FILEIO.
```

```
                :

        CALL        .s_fsinit,A2
          ENDC
        *           Initialize Signal processor
          IFNE  SIGNAL
        CALL        .s_siginit,A2
          ENDC
          IFEQ  SIGNAL
        *       vector all signals to a dumy. routine
        MOVEQ.L     #NUMSIG-1,D0
        GETADDR     .sigvect,A0
        GETADDR     defhndlr,A1
        LL:
        MOVE.L      A1,(A0)+
        DBF         D0,LL
          ENDC
        *           Call main
        GETADDR     .argv,A0
        MOVE.L      A0,-(SP)              ; Pass address of _argv
        GETMEM      .argc,D0,L,A1
        MOVE.L      D0,-(SP)              ; Pass _argc
        CALL        main,A2               ; Call main(argc,argv)
        ADDQ.L      #8,SP
        CLR.L       -(SP)
        CALL        exit,A2               ; exit(0)
          IFEQ  SIGNAL
        *
        *       Default handler for signals
        *
        defhndlr:
        RTS
          ENDC
        *****************************************************************
        *
        *                       Data area
        *
        *****************************************************************
          SECTION   2
        *       Register contents at startup
        .iniregs:
          DS.B  4       D0
          DS.B  4       D1
          DS.B  4       D2
          DS.B  4       D3
          DS.B  4       D4
          DS.B  4       D5
          DS.B  4       D6
          DS.B  4       D7
          DS.B  4       A0
          DS.B  4       A1
          DS.B  4       A2
          DS.B  4       A3
          DS.B  4       A4
          DS.B  4       A5
          DS.B  4       A6
        ENDREGS:
          DS.L TSTKSZ/4                   ; Temp stack for _init()
        .tmpstk:
        *   Storage for command line
```

```
    IFNE    CMDLINE
.cmdline DC.B MAXCMDLN
    ENDC
*       Variables for C runtime library
.maxfiles DS.L 1                        ;   Maximum open files
.file     DS.L 1                        ;   Pointer to file control
errno:
.s_errno  DS.L 1                        ;   Global error number
.argc     DS.L 1                        ;   GLobal argc
.argv     DS.L MAXARG                   ;   Global argv
.hpsize   DS.L 1                        ;   Size of heap segments
.stksize  DS.L 1                        ;   Size of stack segment
.stkaddr  DS.L 1                        ;   Address of stack segment
.sigvect  DS.L NUMSIG
.next_rand DS.L 1                       ;   Random number seed
    SECTION      3
*       beginning of bss (uninitialized) data area
.bssbeg   DC.L  0
    SECTION      4
*       end of bss (uninitialized) data area
.bssend   DC.L  0
    SECTION      14
*       beginning of position independant initializer list
.beginit  DC.L  0
    SECTION      15
*       end of code segment
.endcode  DC.L  0
    END
```

Appendix E

GAUSS MACHINE SCHEMATICS
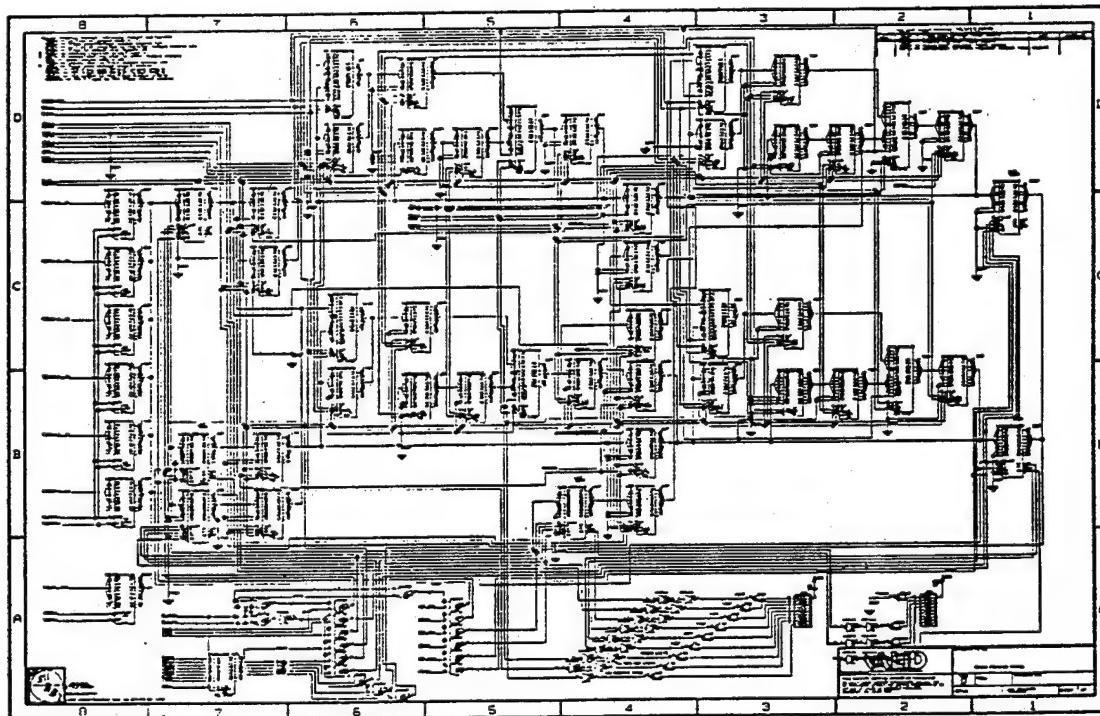

Figure E.1: Gauss Array

244
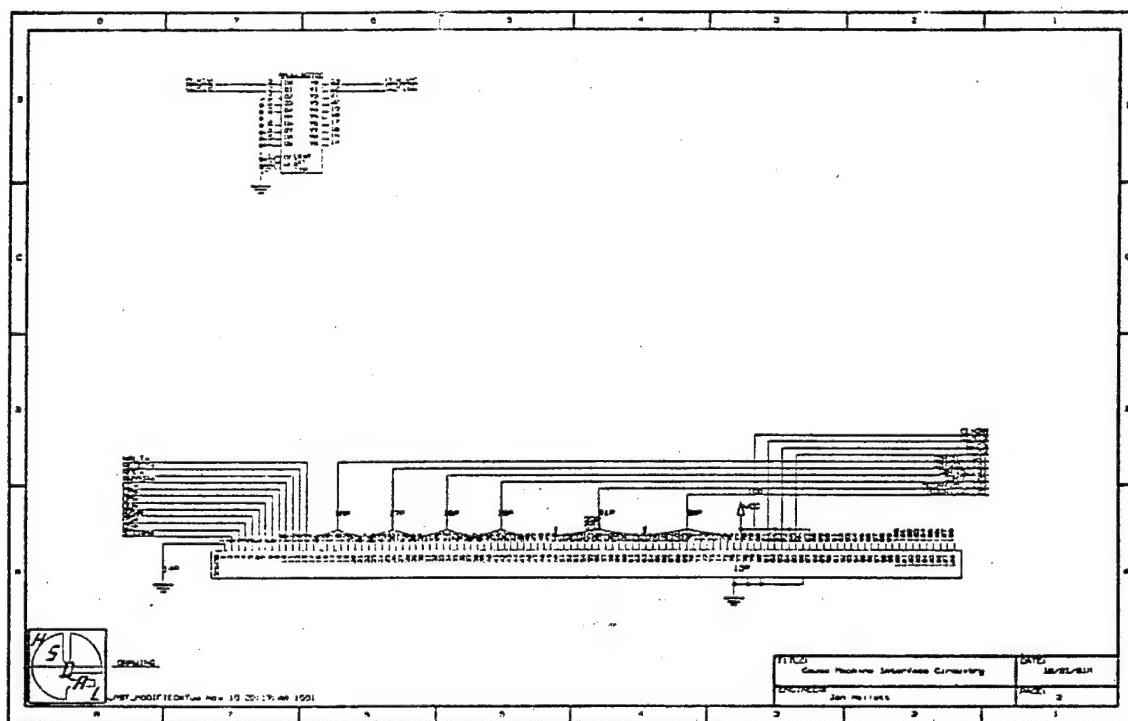
Figure E.2: Gauss Array Miscellaneous
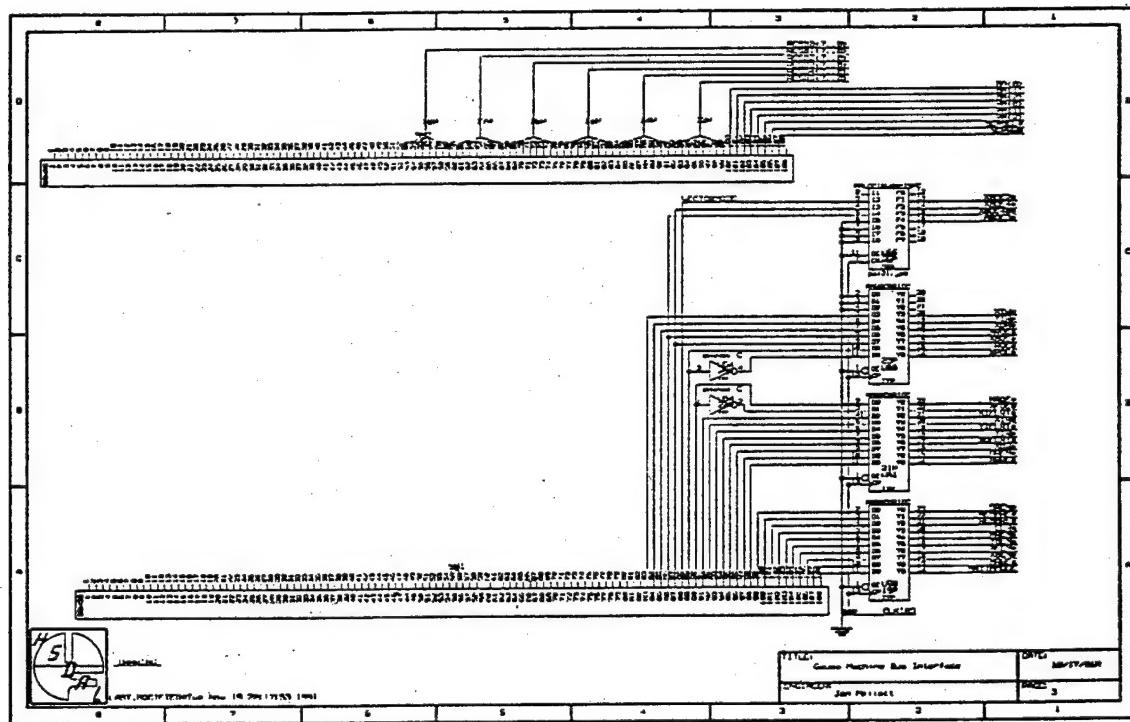
Figure E.3: Gauss Array Instruction Decoding

Appendix F

GAUSS MACHINE PROGRAMMABLE LOGIC DEVICE LISTINGS

This appendix contains listings for the Gauss machine controller's PLDs. These PLDs are used for decoding instructions from the microsequencer, and interface glue between the InvestiGATOR and the controller.

F.1 PAL Listings

F.1.1 PALC1.PDS

```
TITLE    GAUSS MACHINE INSTRUCTION DECODER 1
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;------------------------------- PIN Declarations ----------------
PIN  1         CLK          COMBINATORIAL      ; CLOCK
PIN  2..9      INST[1..8]    COMBINATORIAL      ; INPUT
PIN  10        GND                             ; GROUND
PIN  12        ARITHMODE     REGISTERED         ; OUTPUT
PIN  13        /MRWE         REGISTERED         ; OUTPUT
PIN  14        /MROE         REGISTERED         ; OUTPUT
PIN  15        /XBOE         REGISTERED         ; OUTPUT
PIN  16        /XBEN         REGISTERED         ; OUTPUT
PIN  17        /YBEN         REGISTERED         ; OUTPUT
PIN  19        /XFEN         REGISTERED         ; OUTPUT
PIN  20        VCC                             ; SUPPLY
STRING NOP '/INST[1]*/INST[2]*/INST[3]*/INST[4]*/INST[5]*/INST[6]*/INST[7]*/INST[8]'

EQUATIONS
ARITHMODE = GND
MRWE = GND
MROE = GND
XBOE = GND
XBEN = GND
YBEN = GND
XFEN = GND
```

247

F.1.2 PALC2.PDS

```
TITLE     GAUSS MACHINE INSTRUCTION DECODER 2
PATTERN   A
REVISION  1.0
AUTHOR    JON MELLOTT
COMPANY   HSDAL
DATE      2/23/92

CHIP  DECODER  PALCE16V8

;-------------------------------- PIN Declarations ----------------
PIN   1          CLK          COMBINATORIAL        ; CLOCK
PIN   2..9       INST[1..8]   COMBINATORIAL        ; INPUT
PIN   10         GND                               ; GROUND
PIN   12         /ROE         REGISTERED           ; OUTPUT
PIN   13         /PREN        REGISTERED           ; OUTPUT
PIN   14         /CLR         REGISTERED           ; OUTPUT
PIN   15         /RESBWE      REGISTERED           ; OUTPUT
PIN   16         /RESBRE      REGISTERED           ; OUTPUT
PIN   17         /SREN        REGISTERED           ; OUTPUT
PIN   18         /ARWE        REGISTERED           ; OUTPUT
PIN   19         /AREN        REGISTERED           ; OUTPUT
PIN   20         VCC                               ; SUPPLY
STRING NOP '/INST[1]*/INST[2]*/INST[3]*/INST[4]*/INST[5]*/INST[6]*/INST[7]*/INST[8]'

EQUATIONS
ROE = GND
PREN = GND
CLR = GND
RESBWE = GND
RESBRE = GND
SREN    = GND
ARWE = GND
AREN = GND
```

F.1.3 PALC3.PDS

```
TITLE     GAUSS MACHINE INSTRUCTION DECODER 3
PATTERN   A
REVISION  1.0
AUTHOR    JON MELLOTT
COMPANY   HSDAL
DATE      2/23/92

CHIP  DECODER  PALCE16V8

;-------------------------------- PIN Declarations ----------------
PIN   1          CLK          COMBINATORIAL        ; CLOCK
PIN   2..9       INST[1..8]   COMBINATORIAL        ; INPUT
PIN   10         GND                               ; GROUND
PIN   12         /X0Z         REGISTERED           ; OUTPUT
PIN   13         /X0FLRT      REGISTERED           ; OUTPUT
PIN   14         /Y1R         REGISTERED           ; OUTPUT
PIN   15         /Y1FLRT      REGISTERED           ; OUTPUT
PIN   16         /X1R         REGISTERED           ; OUTPUT
PIN   17         /X1FLRT      REGISTERED           ; OUTPUT
PIN   18         /ARW         REGISTERED           ; OUTPUT
PIN   19         /X0R         REGISTERED           ; OUTPUT
PIN   20         VCC                               ; SUPPLY
STRING NOP '/INST[1]*/INST[2]*/INST[3]*/INST[4]*/INST[5]*/INST[6]*/INST[7]*/INST[8]'

EQUATIONS
```

```
XOW = GND
XOFLRT = GND
YIR = GND
YIFLRT = GND
XIR = GND
XIFLRT = GND
AROE = GND
XOR = GND
```

F.1.4 PALCI.PDS

```
TITLE    GAUSS MACHINE INSTRUCTION DECODER 4
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;------------------------------ PIN Declarations ---------------
PIN  1        CLK          COMBINATORIAL       ; CLOCK
PIN  2..9  -  INST[1..8]   COMBINATORIAL       ; INPUT
PIN  10       GND                              ; GROUND
PIN  12       /YIW         REGISTERED          ; OUTPUT
PIN  13       /XIW         REGISTERED          ; OUTPUT
PIN  14       XTDA         REGISTERED          ; OUTPUT
PIN  15       XTDB         REGISTERED          ; OUTPUT
PIN  16       VECTORMODE   REGISTERED          ; OUTPUT
PIN  17       X1           REGISTERED          ; OUTPUT
PIN  18       X2           REGISTERED          ; OUTPUT
PIN  20       VCC                              ; SUPPLY

STRING NOP  '/INST[1]*/INST[2]*/INST[3]*/INST[4]*/INST[5]*/INST[6]*/INST[7]*/INST[8]'

EQUATIONS
YIW = GND
XIW = GND
XTDA = GND
XTDB = GND
VECTORMODE = GND
X1 = GND
X2 = GND
```

F.1.5 PALC5.PDS

```
TITLE    GAUSS MACHINE ADDRESS DECODER
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;------------------------------ PIN Declarations ---------------
PIN  1        CLK          COMBINATORIAL       ; CLOCK
PIN  2        /ARTSP       COMBINATORIAL       ; INPUT
PIN  3..9     ADDR[14..8]  COMBINATORIAL       ; INPUT
PIN  10       GND                              ; GROUND
PIN  11       /AE          COMBINATORIAL       ; INPUT
PIN  19       /GAUSSP      COMBINATORIAL       ; OUTPUT
```

```
PIN  20           VCC                                           ; SUPPLY
EQUATIONS
GAUSSP=ARYSP*AS*/ADDR[14]*/ADDR[13]*/ADDR[12]*/ADDR[11]*/ADDR[10]*
       /ADDR[9]*/ADDR[8]
```

F.1.6 PALC6.PDS

```
TITLE    GAUSS MACHINE SCP ACCESS CTRL
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;-------------------------------- PIN Declarations ---------------
PIN  1            CLK            COMBINATORIAL       ; CLOCK
PIN  2            /GAUSSP        COMBINATORIAL       ; INPUT
PIN  9            /ARYNA         COMBINATORIAL       ; INPUT
PIN  10           GND                                ; GROUND
PIN  12           /STERM         REGISTERED          ; OUTPUT
PIN  19           /SCPAC         REGISTERED          ; OUTPUT
PIN  20           VCC                                ; SUPPLY

EQUATIONS

STERM := GND
SCPAC := GND
```

F.1.7 PALC7.PDS

```
TITLE    GAUSS MACHINE PORT ADDRESS DECODER
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;-------------------------------- PIN Declarations ---------------
PIN  1            CLK            COMBINATORIAL       ; CLOCK
PIN  2            /ARYNA         COMBINATORIAL       ; INPUT
PIN  4..9         ADDR[7..2]     COMBINATORIAL       ; INPUT
PIN  10           GND                                ; GROUND
PIN  14           BA2            REGISTERED          ; OUTPUT
PIN  15           BA1            REGISTERED          ; OUTPUT
PIN  16           BA0            REGISTERED          ; OUTPUT
PIN  17           PA2            REGISTERED          ; OUTPUT
PIN  18           PA1            REGISTERED          ; OUTPUT
PIN  19           PA0            REGISTERED          ; OUTPUT
PIN  20           VCC                                ; SUPPLY

EQUATIONS
BA2=GND
BA1=GND
BA0=GND
PA2=GND
PA1=GND
PA0=GND
```

F.1.8 PALC8.PDS

```
TITLE    GAUSS MACHINE DATA BUFFER CTRL1
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;-------------------------------- PIN Declarations ----------------
PIN  1           CLK          COMBINATORIAL      ; CLOCK
PIN  2           /ARYNA       COMBINATORIAL      ; INPUT
PIN  3           /GAUSSP      COMBINATORIAL      ; INPUT
PIN  4..6        ADDR[7..5]   COMBINATORIAL      ; INPUT
PIN  7           RW           COMBINATORIAL      ; INPUT
PIN  10          GND                             ; GROUND
PIN  16          /SCPOER1     REGISTERED         ; OUTPUT
PIN  17          /SCPOER2     REGISTERED         ; OUTPUT
PIN  18          /SCPOER3     REGISTERED         ; OUTPUT
PIN  19          /SCPOER4     REGISTERED         ; OUTPUT
PIN  20          VCC                             ; SUPPLY

EQUATIONS
SCPOER1=GND
SCPOER2=GND
SCPOER3=GND
SCPOER4=GND
```

F.1.9 PALC9.PDS

```
TITLE    GAUSS MACHINE DATA BUFFER CTRL1
PATTERN  A
REVISION 1.0
AUTHOR   JON MELLOTT
COMPANY  HSDAL
DATE     2/23/92

CHIP  DECODER  PALCE16V8

;-------------------------------- PIN Declarations ----------------
PIN  1           CLK          COMBINATORIAL      ; CLOCK
PIN  2           /ARYNA       COMBINATORIAL      ; INPUT
PIN  3           /GAUSSP      COMBINATORIAL      ; INPUT
PIN  4..6        ADDR[7..5]   COMBINATORIAL      ; INPUT
PIN  7           RW           COMBINATORIAL      ; INPUT
PIN  10          GND                             ; GROUND
PIN  12          /SCPOER5     REGISTERED         ; OUTPUT
PIN  13          /SCPOER6     REGISTERED         ; OUTPUT
PIN  14          /SCPOET1     REGISTERED         ; OUTPUT
PIN  15          /SCPOET2     REGISTERED         ; OUTPUT
PIN  16          /SCPOET3     REGISTERED         ; OUTPUT
PIN  17          /SCPOET4     REGISTERED         ; OUTPUT
PIN  18          /SCPOET5     REGISTERED         ; OUTPUT
PIN  19          /SCPOET6     REGISTERED         ; OUTPUT
PIN  20          VCC                             ; SUPPLY

EQUATIONS
SCPOER5=GND
SCPOER6=GND
SCPOET1=GND
SCPOET2=GND
SCPOET3=GND
SCPOET4=GND
```

```
SCPOET5=GND
SCPOET6=GND
```

Appendix G

GAUSS MACHINE MICROCODE

This appendix lists the Gauss machine microcode. This microcode is written for Advanced Microdevices' ASM11x microcode assembler. This microcode assembler emits object code for AMD's Am29PLH4x/15x family of EPROM based microsequencers. The listing in Section G.1 is the source for the microcode. Section G.2 describes the microcode and microinstructions.

G.1 Gauss Machine Microcode Listing

```
DEVICE (CPL154)
"Gauss Machine Controller Microsequencer"
"J. Mellott"
"2/24/92"
"High Speed Digital Architecture Laboratory"
"University of Florida"
DEFINE
        "Test inputs"
        SCPACCESS = T7

        "Output control bits"
        STANDBY  = 100#H
        SCPSTDBY = O#H
        ;

BEGIN
"standby and wait for something to happen..."
STDBY:   STANDBY.      IF (SCPACCESS) THEN GOTO PL(SCPACC) ELSE WAIT;
"The SCSI control processor is accessing the array so standby..."
SCPACC:  SCPSTDBY.     IF (NOT SCPACCESS) THEN GOTO PL(STDBY) ELSE WAIT;
END.
```

G.2 Gauss Machine Microcode Description

Appendix II

MACINTOSH API SOURCE CODE

This appendix contains source code for the Macintosh/Gauss machine API.

II.1 TYPES.H

```
/* types.h */
#ifndef __types_h
#define __types_h
#define INTTYPE long            /* integer data type */
#define FLOATTYPE double        /* floating-point data type */
#define SATMODE 1    /* defined for saturated fixed-point arithmetic */
#define NOERR 0      /* OK return code for int function */
/* flags for the type field in a matrix struct */
#define CMPLX 0x1
#define REAL 0x2
 /* Data structure for matrices */
typedef struct {
  int type;                    /* CMPLX or REAL */
  int rows, cols;              /* dimensions */
  FLOATTYPE *re;               /* real part */
  FLOATTYPE *im;               /* imaginary part */
} matrix;
typedef struct {
  int type;                    /* CMPLX or REAL */
  int rows, cols;              /* dimensions */
  INTTYPE *re;                 /* integer real part */
  INTTYPE *im;                 /* integer imag part */
} int_matrix;
/* Labview data structure */
typedef struct {
long dimSizes[2];
double argi[];
} TD1;
typedef TD1 **TD1H41;
#endif
```

II.2 CONV.H

```
/* conv.h */
#ifndef __CONV_H
#define __CONV_H
void init_conv(int wlen, int fbits);
INTTYPE float2fixed(FLOATTYPE f);
FLOATTYPE fixed2float(INTTYPE i);
```

```
/* Converts a floating-point matrix to an integer matrix */
int_matrix *mfloat2fixed(matrix *mat);
/* Converts an integer matrix to a floating-point matrix */
matrix *mfixed2float(int_matrix *mat);
void set_fname(char *fname);
INTTYPE check(INTTYPE result);
#define max(a, b) (a > b) ? a : b /* maximum of a and b */
#define min(a, b) (a < b) ? a : b /* minimum of a and b */
#endif /* __CONV_H */
```

H.3 CONV.C

```
/* conv.c */
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "types.h"
#include "utils.h"
#include "conv.h"
#include "matrix.h"
#include "int_matrix.h"
static long wordlen;
static long fracbits;
static long maxint;
static char func_name[255] = ""; /* current function name, used to report errors */
static int err_count = 0; /* number of errors in current function */
void init_conv(int wlen, int fbits)
    /* int wlen      wordlength   */
    /* int fbits     binary point */
{
  if (wordlen <= 32)              /* max wordlength = 32 bits */
    wordlen = wlen;
  else
    wordlen = 32;
  if (fbits <= wordlen)           /* binary point should be < wordlength */
    fracbits = fbits;
  else
    fracbits = wordlen;
  maxint = (long)(ldexp(1.0, wordlen) - 2); /* maximal absolute value */
}
INTTYPE float2fixed(FLOATTYPE f)
{
  long tmp1, tmp2;
  int exp;
  double mant;
  mant = frexp(f, &exp);          /* split f into mantissa and exponent */
  tmp1 = (long) ldexp(mant, exp + fracbits);  /* mant * 2^(exp + fracbits) */
  tmp2 = tmp1 % maxint;
  if (tmp1 != tmp2)
    {
      error("float2fixed: Overflow"); /* report overflow */
    }
  return((INTTYPE)tmp2);
}
FLOATTYPE fixed2float(INTTYPE i)
{
  double mant;
  mant = (double) i;              /* cast i to double */
  mant /= ldexp(1.0, fracbits);   /* divide by 2^(fracbits) */
  return((FLOATTYPE) mant);
}
```

```c
/* Converts a floating point matrix to an integer matrix */
int_matrix *mfloat2fixed(matrix *mat)
{
  int i;
  int_matrix *tmp;
  if (mat != NULL)
    {
      if ((tmp = int_new_temp(mat->rows, mat->cols, mat->type)) != NULL)
        {
          for (i = 0; i < SIZE(tmp); i++)
              tmp->re[i] = float2fixed(mat->re[i]);
          if (tmp->type == CMPLX)
            {
              for (i = 0; i < SIZE(tmp); i++)
                tmp->im[i] = float2fixed(mat->im[i]);
            }
        }
    }
  return (tmp);
}
/* Converts an integer matrix to a floating-point matrix */
matrix *mfixed2float(int_matrix *mat)
{
  int i;
  matrix *tmp;
  if (mat != NULL)
    {
      if ((tmp = new_temp(mat->rows, mat->cols, mat->type)) != NULL)
        {
          for (i = 0; i < SIZE(tmp); i++)
              tmp->re[i] = fixed2float(mat->re[i]);
          if (tmp->type == CMPLX)
            {
              for (i = 0; i < SIZE(tmp); i++)
                tmp->im[i] = fixed2float(mat->im[i]);
            }
        }
    }
  return (tmp);
}
void set_fname(char *fname)
{
err_count = 0;
strcpy(func_name, fname);
}
INTTYPE check(INTTYPE result)
{
if ((result > maxint || result < -maxint) && err_count == 0)
{
error(strcat(func_name, ": overflow/underflow."));
err_count++;
}
return(result % maxint);
}
```

II.4 INTMATRIX.H

```c
/*********** int_matrix.h ******************/
/* Contains declarations for matrix. */
#ifndef __int_matrix_h
#define __int_matrix_h
```

```
/*  int int_cmplx_promote(int_matrix *a, int_matrix *b)
description:
Promotes, if necessary, the operands a and b to complex matrices, that is,
if either a or b is complex then both a and b are converted to complex matrices.
arguments:
int_matrix *a, *b = matrices to be promoted
returns:
int =  0  if successful
-1 if malloc failed
usage:
res = int_cmplx_promote(a, b); // complex promotes a and b, OK if res == 0
see also:
int_op_check()
*/
int int_cmplx_promote(int_matrix *a, int_matrix *b);

/* int int_op_check(int_matrix *a, int_matrix *b)
description:
Check field type and dimensions. First a and b are complex promoted by
int_cmplx_promote(). Returns COMP if a and b have the same dimensions.
Returns COMP | SCAL if either a or b is a scalar. In addition, if a and b
are complex, CMPLX is ORed to the returned value otherwise CMPLX is ORed to
the returned value. Thus, if type = int_op_check(a,b), then (type & CMPLX) will
be true if either a or b is complex, (type & REAL) will be true if both a and
b are real, (type & COMP) will be true if dimensions match and if a or be is
a scalar, (type & SCAL) will be true if a or b is a scalar.
arguments:
int_matrix *a, *b = matrices to be checked
returns:
int = whose bits are set according to the description above.
usage:
type = int_op_check(a, b); // checks dimensions of a, b. Bits will be set in type
// according to the description above
see also:
int_cmplx_promote(). int_mul_check()
note:
    int_op_check() is used by int_add(), int_pmul(), int_pdiv()
*/
int int_op_check(int_matrix *a, int_matrix *b);

/* int int_mul_check(int_matrix *a, int_matrix *b)
description:
Check field type and dimensions. First a and b are complex promoted by
int_cmplx_promote(). Returns COMP if a and b have the same dimensions.
Returns COMP | SCAL if either a or b is a scalar. In addition, if a and b
are complex, CMPLX is ORed to the returned value otherwise CMPLX is ORed to
the returned value. Thus, if type = int_op_check(a,b), then (type & CMPLX) will
be true if either a or b is complex, (type & REAL) will be true if both a and
b are real. (type & COMP) will be true if dimensions match and if a or be is
a scalar, (type & SCAL) will be true if a or b is a scalar.
arguments:
int_matrix *a, *b  matrices to be checked
returns:
int whose bits are set according to the description above.
usage:
type = int_mul_check(a, b); // checks dimensions of a, b. Bits will be set in type
// according to the description above
see also:
int_cmplx_promote(). int_op_check()
note:
    int_mul_check() is used by int_mul()
*/
int int_mul_check(int_matrix *a, int_matrix *b);

/****************** Memory management of matrices *******************/
/* int_matrix *int_new_matrix(int rows, int cols, int type)
description:
```

```
Allocates memory for a matrix with dimension rows and cols and of type type.
Note that it is the user's responsibility to free this matrix. Use int_new_temp()
to get a temporary matrix (which will be freed by kill_temp_list()
or close_GM()).
arguments:
int rows, cols  dimensions of matrix to be allocated
int type type of matrix, CMPLX for a complex matrix and REAL for a
        real matrix
returns:
int_matrix *  matrix of the requested size and type. NULL if out of memory.
usage:
mat = int_new_matrix(3, 5, CMPLX);  // Allocate memory for a 3x5 complex matrix
see also:
int_copy_matrix(), int_new_temp(), int_copy_temp(), int_kill_matrix(),
kill_temp_list()
note:
It is the user's responsibility to free any matrix that has been allocated with
int_new_matrix() (with int_kill_matrix()).
*/
int_matrix *int_new_matrix(int rows, int cols, int type);
/* void int_kill_matrix(int_matrix *m)
description:
Frees memory used by m. Note that m should have been allocated with int_new_matrix()
or int_copy_matrix().
If m is a temporary matrix, that is, allocated explicitly or implicitly with
int_new_temp() or int_copy_temp(), then kill_temp_list() should be used instead of
int_kill_matrix().
arguments:
int_matrix *m Matrix to be freed, must have been allocated with int_new_matrix()
     or int_copy_matrix()
returns:
nothing
usage:
int_kill_matrix(A);  // Free memory allocated for A
see also:
int_copy_matrix(), int_new_temp(), int_copy_temp(), int_new_matrix(), kill_temp_list()
note:
m must have been allocated explicitly or implicitly with int_new_matrix()
or int_copy_matrix()
*/
void int_kill_matrix(int_matrix *m);
/* int_matrix *int_copy_matrix(int_matrix *source)
description:
Returns a copy of the matrix source. The copy is allocated with int_new_matrix().
Note that it is the user's responsibility to free this matrix. Use int_copy_temp()
to get a temporary matrix (which will be freed by kill_temp_list()
or close_GM()).
arguments:
int_matrix *source  matrix to be copied
returns:
int_matrix *  copy of source. NULL if out of memory.
usage:
new = int_copy_matrix(old);  // copy the matrix old to the matrix new
see also:
int_kill_matrix(), int_new_temp(), int_copy_temp(), int_new_matrix(), kill_temp_list()
note:
It is the user's responsibility to free any matrix that has been allocated with
int_copy_matrix() (with int_kill_matrix()).
*/
int_matrix *int_copy_matrix(int_matrix *source);
/* int_matrix *int_new_temp(int rows, int cols, int type)
description:
Allocates memory for a temporary matrix with dimension rows and cols and
of type type.
Note that it is the user's responsibility to free this matrix. To free ALL
temporary matrices, use kill_temp_list() or close_GM().
```

```
arguments:
int rows, cols  dimensions of matrix to be allocated
int type type of matrix. CMPLX for a complex matrix and REAL for a
        real matrix
returns:
int_matrix *  matrix of the requested size and type. NULL if out of memory.
usage:
mat = int_new_temp(3, 5, CMPLX);// Allocate memory for a temporary 3x5 complex matrix
see also:
int_kill_matrix(). int_copy_matrix(). int_copy_temp(), int_new_matrix(),
kill_temp_list()
*/
int_matrix *int_new_temp(int rows, int cols, int type);

/* int_matrix *int_copy_temp(int_matrix *source)
description:
Returns a copy of the matrix source. The copy is allocated with int_new_temp()
and therfore, is a temporary matrix.
To free ALL temporary matrices, use kill_temp_list() or close_GM().
arguments:
int_matrix *source  matrix to be copied
returns:
int_matrix *  copy of source. NULL if out of memory.
usage:
new = int_copy_temp(old);  // copy the matrix old to the temporary matrix new
see also:
int_kill_matrix(), int_copy_matrix(). int_new_temp(), int_new_matrix(),
kill_temp_list()
*/
int_matrix *int_copy_temp(int_matrix *source);

/*  int_matrix *int_real(int_matrix *mat)
description:
Returns a matrix containing the real part of mat
arguments:
int_matrix *mat  input matrix
returns:
int_matrix * real part of mat
usage:
real_part = int_real(cmplx_matrix); // real_part = Re[cmplx_matrix]
MATLAB equivalent:
>> real_part = real(cmplx_matrix);
*/
int_matrix *int_real(int_matrix *mat);

/*  int_matrix *int_imag(int_matrix *mat)
description:
Returns a matrix containing the imaginary part of mat
arguments:
int_matrix *mat  input matrix
returns:
int_matrix * imaginary part of mat
usage:
im_part = int_imag(cmplx_matrix); // im_part = Im[cmplx_matrix]
MATLAB equivalent:
>> im_part = imag(cmplx_matrix);
*/
int_matrix *int_imag(int_matrix *mat);

/*  int int_max_index(int_matrix *mat)
description:
Return index to maximal element of mat
arguments:
int_matrix *mat  input matrix
returns:
int index to the maximal element of mat
usage:
max_i = int_max_index(mat); // max(mat) = mat[max_i]
MATLAB equivalent
>> max_i = find(mat == max(mat:));
```

```
*/
int int_max_index(int_matrix *mat);
/*  int int_min_index(int_matrix *mat)
description:
Return index to minimal element of mat
arguments:
int_matrix *mat   input matrix
returns:
int index to the minimal element of mat
usage:
min_i = int_min_index(mat); // min(mat) = mat[min_i]
MATLAB equivalent:
>> min_i = find(mat == min(mat(:)));
*/
int int_min_index(int_matrix *mat);
/*  int_matrix *int_minus(int_matrix *mat)
description:
Negates elements of mat
arguments:
int_matrix *mat   input matrix
returns:
int_matrix * the negated input matrix
usage:
minus_A = int_minus(A); // minus_A = -A
MATLAB equivalent:
>> minus_A = -A;
*/
int_matrix *int_minus(int_matrix *mat);
/*  int_matrix *int_conj(int_matrix *mat)
description:
conjugates elements of mat
arguments:
int_matrix *mat   input matrix
returns:
int_matrix * the conjugated input matrix
usage:
conj_A = int_conj(A); // conj_A = -A
MATLAB equivalent:
>> conj_A = int_conj(A);
*/
int_matrix *int_conj(int_matrix *mat);
/*  int_matrix *int_add(int_matrix *a, int_matrix *b)
description:
adds matrices a and b
arguments:
int_matrix *a, b   input matrices
returns:
int_matrix * the sum of a and b, NULL if error
usage:
sum = int_add(a, b); // sum = a + b
MATLAB equivalent:
>> sum = a + b;
*/
int_matrix *int_add(int_matrix *a, int_matrix *b);
/*  int_matrix *int_pmul(int_matrix *a, int_matrix *b)
description:
pointwise multiplication of matrices a and b
arguments:
int_matrix *a, b   input matrices
returns:
int_matrix * the pointwise product of a and b, NULL if error
usage:
pprod = int_pmul(a, b); // pprod = a .* b
MATLAB equivalent:
>> pprod = a .* b;
*/
int_matrix *int_pmul(int_matrix *a, int_matrix *b);
```

```
/*  int_matrix *int_mul(int_matrix *a, int_matrix *b)
description:
multiplication of matrices a and b
arguments:
int_matrix *a, b  input matrices
returns:
int_matrix * the product of a and b, NULL if error
usage:
prod = int_mul(a, b); // prod = a * b
MATLAB equivalent:
>> prod = a * b;
*/
int_matrix *int_mul(int_matrix *a, int_matrix *b);
/*  int_matrix *int_appendrows(int_matrix *a, int_matrix *b)
description:
appends b's rows to a
arguments:
int_matrix *a, b  input matrices
returns:
int_matrix * [a; b], NULL if error
usage:
c = int_appendrows(a, b); // c = [a; b]
MATLAB equivalent:
>> c = [a; b];
*/
int_matrix *int_appendrows(int_matrix *a, int_matrix *b);
/*  int_matrix *int_appendcols(int_matrix *a, int_matrix *b)
description:
appends b's columns to a
arguments:
int_matrix *a, b  input matrices
returns:
int_matrix * [a, b], NULL if error
usage:
c = int_appendcols(a, b); // c = [a, b]
MATLAB equivalent:
>> c = [a, b];
*/
int_matrix *int_appendcols(int_matrix *a, int_matrix *b);
/*  int_matrix *int_transp(int_matrix *mat)
description:
transposes elements of mat. Note: does not conjugate elements. Use int_herm()
for conjugate transpose (hermitian).
arguments:
int_matrix *mat  input matrix
returns:
int_matrix * the transposed input matrix, NULL if error.
usage:
tran_A = int_transp(A); // tran_A = A'
MATLAB equivalent:
>> A = A.'; % Note: ".'" not "'", that is, int_conj() is not conjugate transpose
see also:
int_herm()
*/
int_matrix *int_transp(int_matrix *mat);
/*  int_matrix *int_herm(int_matrix *mat)
description:
conjugate transposes elements of mat, that is, takes the hermitian of mat.
arguments:
int_matrix *mat  input matrix
returns:
int_matrix * the conjugate transposed input matrix, NULL if error.
usage:
tran_A = int_transp(A); // tran_A = A'
MATLAB equivalent:
>> A = A'; % Note: "'" not ".'", that is, int_herm() is conjugate transpose
```

```
see also:
int_transp()
*/
int_matrix *int_herm(int_matrix *mat);

/*  int_matrix *int_index_rows(int_matrix *mat, int_matrix *ind)
description:
Returns the rows of mat that are pointed out by ind. The elements of ind are
used to pick out rows. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3 1], then the result
would be a matrix of the form [7 8 9; 4 5 6]. This is analoguous to the MATLAB
statement, mat(ind, :).
arguments:
int_matrix *mat   matrix to be indexed
int_matrix *ind   row indexing matrix
returns:
int_matrix * the indexed input matrix, NULL if error.
usage:
B = int_index_rows(mat, ind); // B = mat(ind, :)
MATLAB equivalent:
>> B = mat(ind, :)
see also:
int_index_cols(), int_index_rows_cols(), int_sub_matrix()
note:
For greatest convenience, use int_sub_matrix() for all indexing purposes.
*/
int_matrix *int_index_rows(int_matrix *mat, int_matrix *ind);

/*  int_matrix *int_index_cols(int_matrix *mat, int_matrix *ind)
description:
Returns the columns of mat that are pointed out by ind. The elements of ind are
used to pick out columns. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3 1], then the result
would be a matrix of the form [3 1; 6 5; 9 8]. This is analoguous to the MATLAB
statement, mat(:, ind).
arguments:
int_matrix *mat   matrix to be indexed
int_matrix *ind   column indexing matrix
returns:
int_matrix * the indexed input matrix, NULL if error.
usage:
B = int_index_cols(mat, ind); // B = mat(:, ind)
MATLAB equivalent:
>> B = mat(:, ind)
see also:
int_index_rows(), int_index_rows_cols(), int_sub_matrix()
note:
For greatest convenience, use int_sub_matrix() for all indexing purposes.
*/
int_matrix *int_index_cols(int_matrix *mat, int_matrix *ind);

/*  int_matrix *int_index_rows_cols(int_matrix *mat, int_matrix *ind_a, int_matrix *ind_b)
description:
Returns the rows and columns of mat that are pointed out by ind_a and ind_b.
The elements of ind_a and ind_b
are used to pick out rows and columns, respectively. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind_a = [3 1] and ind_b = [2],
then the result would be a matrix of the form [8; 2] (that is, elements (3,2)
and (1,2)). This is analoguous to the MATLAB statement, mat(ind_a, ind_b).
arguments:
int_matrix *mat   matrix to be indexed
int_matrix *ind_a  row indexing matrix
int_matrix *ind_b  column indexing matrix
returns:
int_matrix * the indexed input matrix, NULL if error.
usage:
B = int_index_rows_cols(mat, ind_a, ind_b); // B = mat(ind_a, ind_b)
MATLAB equivalent:
>> B = mat(ind_a, ind_b)
```

```
see also:
int_index_rows(), int_index_cols(), int_sub_matrix()
note:
For greatest convenience, use int_sub_matrix() for all indexing purposes.
*/
int_matrix *int_index_rows_cols(int_matrix *mat, int_matrix *ind_a, int_matrix *ind_b);
/*  int_matrix *int_sub_matrix(int_matrix *mat, int_matrix *rows, int_matrix *cols);
description:
Returns the rows and columns of mat that are pointed out by (the matrices)
rows and cols.
The elements of rows and cols
are used to pick out rows and columns, respectively. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and rows = [3.14 1.99] and cols = [2,1],
then the result would be a matrix of the form [8; 2] (that is, elements (3,2)
and (1,2)). This is analoguous to the MATLAB statement, mat(rows, cols).
To just index rows, like MATLAB's mat(rows, :), set cols to NULL. Similary,
to just index columns, like MATLAB's mat(:, cols), set rows to NULL. By using
this scheme all indexing can be done with int_sub_matrix().
arguments:
int_matrix *mat   matrix to be indexed
int_matrix *rows   row indexing matrix
int_matrix *cols   column indexing matrix
returns:
int_matrix *  the indexed input matrix, NULL if error.
usage:
B = int_sub_matrix(mat, rows, cols); // B = mat(rows, cols)
B = int_sub_matrix(mat, rows, NULL); // B = mat(rows, :)
B = int_sub_matrix(mat, NULL, cols); // B = mat(:, cols)
MATLAB equivalent:
>> B = mat(rows, cols)
see also:
int_index_rows(), int_index_cols(), int_index_rows_cols()
note:
For greatest convenience, use int_sub_matrix() for all indexing purposes.
*/
int_matrix *int_sub_matrix(int_matrix *mat, int_matrix *rows, int_matrix *cols);
/*  int_matrix *int_assign(int_matrix *target, int_matrix *rows, int_matrix *cols, int_matrix
description:
Puts the matrix source into a submatrix of target indicated by rows and cols.
That is, rows and cols defines a submatrix of target (exactly like int_sub_matrix())
and this submatrix is overwritten with data from the source matrix. This is
analoguous to the MATLAB statement target(rows, cols) = source. Needless to
say, the submatrix of target and source must be of the same dimensions.
For example, suppose
target = [1  2  3  4;
   5  6  7  8;
   9 10 11 12],
source = [13 14;
          15 16],
    rows = [3 2] and cols = [1 2], the resulting matrix would be
[ 1  2  3  4;
 15 16  7  8;
 13 14 11 12 ].
If rows or cols is NULL, this means all the rows and all the columns of target.
That is, target(rows,:) = source would be coded as
int_assign(target, rows, NULL, source), and similary, target(rows,:) = source
would be coded as int_assign(target, rows, NULL, source).
arguments:
int_matrix *target matrix to be written to
int_matrix *rows row indexing matrix
int_matrix *cols column indexing matrix
int_matrix *source matrix whose data will be written to target.
returns:
int_matrix *  copy of target, NULL if error.
usage:
```

```
int_assign(target, rows, cols, source) // target(rows, cols) = source
int_assign(target, rows, NULL, source) // target(:, cols) = source
int_assign(target, NULL, cols, source) // target(rows, :) = source
see also:
int_sub_matrix() int_copy_temp()
note:
Does not handle the case target(:,:) = source. For this use
target = int_copy_temp(source).
*/
int_matrix *int_assign(int_matrix *target, int_matrix *rows, int_matrix *cols, int_matrix *sou

/*  int_matrix *int_range(INTTYPE from, INTTYPE step, INTTYPE to);
description:
Creates a vector like MATLAB's from:step:to. If step is 0, then it is set to 1.
For example, int_range(1, 2, 7) results in [1 3 5 7], int_range(3, 5) results in
[3 4 5].
arguments:
INTTYPE from start value
INTTYPE step step size
INTTYPE to stop value
returns:
int_matrix * vector with elements starting at from and stopping at to,
spaced by step. NULL if error.
usage:
int_range(from, step, to) // from:step:to
int_range(from, 0, to) // from:to
*/
int_matrix *int_range(INTTYPE from, INTTYPE step, INTTYPE to);

/*  int_matrix *int_scl2mat(INTTYPE re, INTTYPE im, int type);
description:
Creates a 1x1 matrix from the scalar (re + j*im), if type is CMPLX. If type is
REAL the imaginary part is ignored.
arguments:
INTTYPE re real part
INTTYPE im imaginary part
returns:
int_matrix * 1x1 matrix with the element (re + j*im). NULL if error.
usage:
scalar_mat = int_scl2mat(3, 2, CMPLX); //  scalar_mat(1,1) = 3 + j*2
scalar_mat = int_scl2mat(3, 2, REAL);  //  scalar_mat(1,1) = 3
*/
int_matrix *int_scl2mat(INTTYPE re, INTTYPE im, int type);
#endif
```

II.5 INTMATRIX.C

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "types.h"
#include "int_matrix.h"
#include "matrix.h"
#include "list.h"
#include "conv.h"
/* Private declarations */
static LIST *temp_list;          /* list for temporary matrices */
/* stuff used in op_check and mul_check */
#define COMP 0x4                 /* marks compatible dimensions */
#define SCAL 0x8                 /* marks one operand as a scalar */
#define INT(a) ((int)(a))
/* Are a and b of the same dimensions */
```

```c
#define EQDIM(a, b) ( (a->rows == b->rows) &&  (a->cols == b->cols) )
/* Allocates memory for a int_matrix with dimension rows and cols */
int_matrix *int_new_matrix(int rows, int cols, int type)
{
  int_matrix *mat;
  mat = NULL;
  /* Try to allocate int_matrix */
  if ((mat = (int_matrix*)malloc(sizeof(int_matrix))) != NULL)
    {
      mat->cols = cols; mat->rows = rows; mat->type = type;
      /* Try to allocate real part */
      if ((mat->re = (INTTYPE*)malloc(SIZE(mat) * sizeof(INTTYPE))) != NULL)
        {
          if (type == CMPLX)    /* Try to allocate imaginary part */
            if ((mat->im = (INTTYPE*)malloc(SIZE(mat) * sizeof(INTTYPE))) == NULL)
              {                  /* malloc of mat->im failed, cleanup */
                free(mat->re);
                free(mat);
                mat = NULL;
              }
        }
      else                      /* malloc of mat->re failed, cleanup */
        {
          free(mat);
          mat = NULL;
        }
    }
  else                          /* malloc of mat failed, cleanup */
    {
      return(NULL);
    }
  return(mat);
}
/* Returns a copy the integer matrix source */
int_matrix *int_copy_matrix(int_matrix *source)
{
  int_matrix *target;
  INTTYPE *tre, *sre;
  int i;
  if ((target = int_new_matrix(source->rows, source->cols, source->type)) != NULL)
    {
      sre = source->re;
      tre = target->re;
      for (i = 0; i < SIZE(source); i++)
        tre[i] = sre[i];
      if (source->type == CMPLX)
        {
          sre = source->im;
          tre = target->im;
          for (i = 0; i < SIZE(source); i++)
            tre[i] = sre[i];
        }
      return(target);
    }
  return(NULL);
}
/* Allocates memory for a temp int_matrix with dimension rows and cols */
int_matrix *int_new_temp(int rows, int cols, int type)
{
  int_matrix *mat;
  if ((mat = int_new_matrix(rows, cols, type)) != NULL)
    {
      appendlist(temp_list, mat);
    }
```

```
    return(mat);
}
/* Copies int_matrix m to a temp int_matrix */
int_matrix *int_copy_temp(int_matrix *source)
{
    int_matrix *target;
    if ((target = int_copy_matrix(source)) != NULL) /* copy source int_matrix */
    {
        if (appendlist(temp_list, target) == NULL) /* Try to insert target
                                                       in temp_list */
            int_kill_matrix(target);
    }
    return target;                      /* return copy */
}
/* Frees memory used by m */
void int_kill_matrix(int_matrix *m)
{
    if (m != NULL)
    {
        if (m->re != NULL)
            free(m->re);
        if (m->type == CMPLX)
            if (m->im != NULL)
                free(m->im);
    }
}

/* Promotes, if necessary, the operands a and b to complex matrices */
int int_cmplx_promote(int_matrix *a, int_matrix *b)
{
int i;
    if (a != NULL && b != NULL)
    {
        if (a->type == CMPLX && b->type == REAL)
        {
            if ((b->im = (INTTYPE*)malloc(SIZE(b) * sizeof(INTTYPE))) != NULL)
            {
                b->type = CMPLX;
                for (i = 0; i < SIZE(b); i++)
                    b->im[i] = 0;
                return 0;
            }
            else
            {
                return -1;
            }
        }
        if (b->type == CMPLX && a->type == REAL)
        {
            if ((a->im = (INTTYPE*)malloc(SIZE(a) * sizeof(INTTYPE))) != NULL)
            {
                a->type = CMPLX;
                for (i = 0; i < SIZE(a); i++)
                    a->im[i] = 0;
                return 0;
            }
            else
            {
                return -1;
            }
        }
    }
    return 0;
}
int int_mul_check(int_matrix *a, int_matrix *b)
{
```

266

```
      int type = 0;
      int_matrix *tmp;
      if (a != NULL && b != NULL)
        {
          if (a->type == CMPLX || b->type == CMPLX)
            /* Need for complex promotion? */
            {
              int_cmplx_promote(a, b);
              type |= CMPLX;
            }
          else
            type |= REAL;
          if (a->cols == b->rows)
            type |= COMP;             /* a and b are compatible */
          if (SIZE(b) == 1 || SIZE(a) == 1) /* a or b is scalar */
              type |= SCAL | COMP;  /* a and b are compatible */
        }
      return(type);
    }
int int_op_check(int_matrix *a, int_matrix *b)
{
      int type = 0;
      int_matrix *tmp;
      if (a != NULL && b != NULL)
        {
          if (a->type == CMPLX || b->type == CMPLX)
            /* Need for complex promotion? */
            {
              int_cmplx_promote(a, b);
              type |= CMPLX;
            }
          else
            type |= REAL;
          if (a->rows == b->rows && a->cols == b->cols)
            type |= COMP;             /* a and b are compatible */
          if (SIZE(b) == 1 || SIZE(a) == 1) /* a or b is scalar */
              type |= SCAL | COMP;  /* a and b are compatible */ .
        }
      return(type);
    }
/* addition */
int_matrix *int_add(int_matrix *a, int_matrix *b)
{
      int_matrix *sum;
      INTTYPE *sre, *sim, *are, *aim, *bre, *bim;
      int i, type;
      set_fname("int_add");
      sum = NULL;
      if ( (type = int_op_check(a, b)) & COMP)
        {
          /* a and b are compatible */
          if (type & SCAL)            /* either a or b is a scalar */
            {
              if (SIZE(b) == 1)                 /* b is a scalar */
                {
                  are = b->re; aim = b->im; /* interchange a and b */
                  bre = a->re; bim = a->im; /* so scalar is in are/aim   */
                  sum = int_copy_temp(a); /* copy the non-scalar to sum */
                }
              else                      /* a is scalar */
                {
                  are = a->re; aim = a->im;
                  bre = b->re; bim = b->im;
```

```
                    sum = int_copy_temp(b); /* copy the non-scalar to sum */
                }
        }
        else                            /* a and b are non-scalars */
          {
            are = a->re; aim = a->im;
            bre = b->re; bim = b->im;
            sum = int_copy_temp(b);   /* copy b to sum */
          }
        if (sum != NULL)         /* if copy was successful */
          {
            sre = sum->re;
            sim = sum->im;
            if (type & SCAL)        /* scalar addition */
              {
                for (i = 0; i < SIZE(sum); i++)
                  {
                    sre[i] = check(sre[i] + are[0]); /* scalar is in are/aim... */
                    if (type & CMPLX) /* and sre/sim is a copy of non-scalar */
                      sim[i] = check(sim[i] + aim[0]);
                  }
              }
            else
              {
                for (i = 0; i < SIZE(sum); i++)
                  {
                    sre[i] = check(sre[i] + are[i]); /* sre/sim is a copy of b */
                    if (type & CMPLX)
                      sim[i] = check(sim[i] + aim[i]);
                  }
              }
          }
      }
  return(sum);
}
/* pointwise multiplication */
int_matrix *int_pmul(int_matrix *a, int_matrix *b)
{
  int_matrix *prod;
  INTTYPE *pre, *pim, *are, *aim, *bre, *bim;
  int i, type;
  set_fname("int_pmul");
  prod = NULL;
  if ((type = int_op_check(a, b)) & COMP)
    {
      /* a and b are compatible */
      if (type & SCAL)              /* either a or b is a scalar */
        {
          if (SIZE(b) == 1)              /* b is a scalar */
            {
              are = b->re; aim = b->im; /* interchange a and b */
              bre = a->re; bim = a->im; /* so scalar is in are/aim   */
              prod = int_copy_temp(a); /* copy the non-scalar to prod */
            }
          else                            /* a is scalar */
            {
              are = a->re; aim = a->im;
              bre = b->re; bim = b->im;
              prod = int_copy_temp(b); /* copy the non-scalar to prod */
            }
        }
      else                            /* a and b are non-scalars */
        {
          are = a->re; aim = a->im;
          bre = b->re; bim = b->im;
```

```
            prod = int_copy_temp(b);  /* copy b to prod */
        }
      if (prod != NULL)          /* copy was successful */
        {
          pre = prod->re;
          pim = prod->im;
          if (type & SCAL)        /* scalar multiplication.        */
            {                     /* scalar is in are/aim and      */
                                  /* pre/pim is a copy of non-scalar */
              for (i = 0; i < SIZE(prod); i++)
                {
                  if (type & REAL) /* real multiplication */
                    pre[i] = check(pre[i] * are[0]);
                  else          /* complex multiplication */
                    {
                      pre[i] = check(check(are[0] * bre[i]) - check(aim[0] * bim[i]));
                      pim[i] = check(check(are[0] * bim[i]) + check(aim[0] * bre[i]));
                    }
                }
            }
          else                    /* ordinary pointwise multiplication */
            {                     /* pre/pim is a copy of b            */
              for (i = 0; i < SIZE(prod); i++)
                {
                  if (type & REAL) /* real multiplication */
                    pre[i] = check(pre[i] * are[i]);
                  else
                    {             /* complex multiplication */
                      pre[i] = check(check(are[i] * bre[i]) - check(aim[i] * bim[i]));
                      pim[i] = check(check(are[i] * bim[i]) + check(aim[i] * bre[i]));
                    }
                }
            }
        }
    }
  return(prod);
}
/* multiplication */
int_matrix *int_mul(int_matrix *a, int_matrix *b)
{
  int_matrix *prod;
  INTTYPE *pre, *pim, *are, *aim, *bre, *bim, tmp;
  int i, j, k, type;
  set_fname("int_mul");
  prod = NULL;
  if ((type = int_mul_check(a, b)) & COMP)
    {
      /* a and b are compatible */
      if (type & SCAL)          /* either a or b is a scalar */
        {
          if (SIZE(b) == 1)             /* b is a scalar */
            {
              are = b->re; aim = b->im; /* interchange a and b */
              bre = a->re; bim = a->im; /* so scalar is in are/aim */
              prod = int_copy_temp(a); /* copy the non-scalar to prod */
            }
          else
            {                           /* a is scalar */
              are = a->re; aim = a->im;
              bre = b->re; bim = b->im;
              prod = int_copy_temp(b); /* copy the non-scalar to prod */
            }
        }
```

```
    else                      /* a and b are non-scalars */
    {
        are = a->re; aim = a->im;
        bre = b->re; bim = b->im;
        prod = int_new_temp(a->rows, b->cols, a->type);   /* copy b to prod */
    }
    if (prod  != NULL)         /* copy was successful */
    {
        pre = prod->re;
        pim = prod->im;
        if (type & SCAL)        /* scalar multiplication.        */
        {                       /* scalar is in are/aim and       */
                                /* pre/pim is a copy of non-scalar */
            for (i = 0; i < SIZE(prod); i++)
              if (type & REAL) /* real multiplication */
                pre[i] = check(are[0] * bre[i]);
              else
              {                   /* complex multiplication */
                pre[i] = check(check(are[0] * bre[i]) - check(aim[0] * bim[i]));
                pim[i] = check(check(are[0] * bim[i]) + check(aim[0] * bre[i]));
              }
        }
        else                    /* ordinary matrix multiplication */
        {                       /* pre/pim is a copy of b          */
            for (i = 0; i < prod->rows; i++)
            {                   /* for ith row of prod */
              if (type & REAL)
              {                   /* real multiplication */
                for (j = 0; j < prod->cols; j++)/* for jth col of prod */
                {
                    pre[i + j  * prod->rows] = 0;
                    for (k = 0; k < a->cols; k++)
                    {                          /* compute prod(i,j) */
                        pre[i + j * prod->rows] = check(pre[i + j * prod->rows] + check(
                    }
                }
              }
              else
              {                  /* complex multiplication */
                for (j = 0; j < prod->cols; j++)/* for jth col of prod */
                {
                    pre[i + j  * prod->rows] = 0;
                    pim[i + j  * prod->rows] = 0;
                    for (k = 0; k < a->cols; k++)
                    {                          /* compute prod(i,j) */
                        tmp = check(check(are[i + k  * a->rows] * bre[k + j * b->rows])
                                    check(aim[i + k  * a->rows] * bim[k + j * b->rows]))
                        pre[i + j * prod->rows] = check(pre[i + j * prod->rows] + tmp);
                        tmp = check(check(are[i + k  * a->rows] * bim[k + j * b->rows])
                                    check(aim[i + k  * a->rows] * bre[k + j * b->rows]))
                        pim[i + j * prod->rows] =
                            check(pim[i + j * prod->rows] + tmp);
                    }
                }
              }
            }
        }
    }
    return(prod);
}
/* Takes the the real part of m */
```

```
int_matrix *int_real(int_matrix *m)
{
  int_matrix *real;
  if ((real = int_copy_temp(m)) != NULL)
    {
      if (real->type == CMPLX)
        {
          real->type = REAL;
          if (real->im != NULL)
            {
              free(real->im);
              real->im = NULL;
            }
        }
    }
  return(real);
}
/* Takes the the imaginary part of m */
int_matrix *int_imag(int_matrix *m)
{
  int_matrix *imag;
  int i;
  if ((imag = int_copy_temp(m)) != NULL)
    {
      if (imag->type == CMPLX)
        {                            /* if complex get rid of real part */
          if (imag->re != NULL)
              free(imag->re);
          imag->re = imag->im;   /* make old imag part be new real part */
          imag->im = NULL;       /* clean up */
        }
      else
        {                            /* if not complex, the imag part is zero */
          for (i = 0; i < SIZE(imag); i++)
              imag->re[i] = 0;
        }
      imag->type = REAL;         /* the imag part is real */
    }
  return(imag);
}
/* Return index to maximal element of m */
int int_max_index(int_matrix *m)
{
  int i, mx;
  INTTYPE *re, *im;
  mx = 0;
  if (m != NULL)
    {
      re = m->re;
      im = m->im;
      if (m->type = REAL)
        {
          for (i = 1; i < SIZE(m); i++)
            {
              if (re[i] > re[mx])
                  mx = i;
            }
        }
      else
        {
          for (i = 1; i < SIZE(m); i++)
            {
              if (cabs(re[i], im[i]) > cabs(re[mx], im[mx]))
                  mx = i;
            }
```

```
        }
    }
    return(mx);
}
/* Return index to minimal element of m */
int int_min_index(int_matrix *m)
{
    int i, mi;
    INTTYPE *re, *im;
    mi = 0;
    if (m != NULL)
    {
        re = m->re;
        im = m->im;
        if (m->type == REAL)
        {
            for (i = 1; i < SIZE(m); i++)
            {
                if (re[i] < re[mi])
                    mi = i;
            }
        }
        else
        {
            for (i = 1; i < SIZE(m); i++)
            {
                if (cabs(re[i], im[i]) < cabs(re[mi], im[mi]))
                    mi = i;
            }
        }
    }
    return(mi);
}
/* negates m */
int_matrix *int_minus(int_matrix *m)
{
    int i, size;
    int_matrix *neg;
    INTTYPE *re, *im;
    if (m != NULL)
    {
        if ((neg = int_copy_temp(m)) != NULL)
        {
            re = neg->re;
            im = neg->im;
            for (i=0; i < SIZE(m); i++)
                re[i] = -re[i];
            if (m->type == CMPLX)
            {
                for (i=0; i < SIZE(m); i++)
                    im[i] = -im[i];
            }
        }
    }
    return(neg);
}
/* conjugate m */
int_matrix *int_conj(int_matrix *m)
{
    int i;
    int_matrix *conj;
    INTTYPE *im;
    if (m != NULL)
    {
        if ((conj = int_copy_temp(m)) != NULL)
```

```
        {
            im = conj->im;
            if (m->type == CMPLX)
            {
                for (i=0; i < SIZE(m); i++)
                    im[i] = -im[i];
            }
        }
    }
    return(conj);
}
/* append b's rows to a */
int_matrix *int_appendrows(int_matrix *a, int_matrix *b)
{
    int   ai, bi, ci, i, j;
    INTTYPE *cre, *cim, *are, *aim, *bre, *bim;
    int_matrix *c;
    if (b != NULL)
    {
        if (a == NULL)
        {
            return(b);
        }
        if (a->cols == b->cols)
        {
            int_cmplx_promote(a, b);
            if ((c = int_new_temp(a->rows + b->rows, a->cols, a->type)) != NULL)
            {
                are = a->re; aim = a->im;
                bre = b->re; bim = b->im;
                cre = c->re; cim = c->im;
                ai = bi = ci = 0;
                for (j = 0; j < a->cols; j++)
                {
                    for (i = 0; i < a->rows; i++)
                        cre[ci++] = are[ai++];
                    for (i = 0; i < b->rows; i++)
                        cre[ci++] = bre[bi++];
                }
                ai = bi = ci = 0;
                if (a->type == CMPLX)
                    for (j = 0; j < a->cols; j++)
                    {
                        for (i = 0; i < a->rows; i++)
                            cim[ci++] = aim[ai++];
                        for (i = 0; i < b->rows; i++)
                            cim[ci++] = bim[bi++];
                    }
            }
            return(c);
        }
        else
{
// error: dimension mismatch
error("int_appendrows: dimension mismatch");
}
    }
    return(NULL);
}
/* append b's cols to a */
int_matrix *int_appendcols(int_matrix *a, int_matrix *b)
{
    INTTYPE *are, *aim, *bre, *bim, *cre, *cim;
    int i, j;
    int_matrix *c;
```

```
        if (b != NULL)
          {
            if (a == NULL)
              {
                return(b);
              }
            if (a->rows == b->rows)
              {
                if ((c = int_new_temp(a->rows, a->cols + b->cols, a->type)) != NULL)
                  {
                    int_cmplx_promote(a, b);
                    are = a->re; aim = a->im;
                    bre = b->re; bim = b->im;
                    cre = c->re; cim = c->im;
                    for (i = 0; i < SIZE(a); i++)
                      cre[i] = are[i];
                    for (j = 0; j < SIZE(a); j++)
                      cre[i + j] = bre[j];
                    if (a->type == CMPLX)
                      {
                        for (i = 0; i < SIZE(a); i++)
                          cim[i] = aim[i];
                        for (j = 0; j < SIZE(a); j++)
                          cim[i + j] = bim[j];
                      }
                  }
                return(c);
              }
        else
        {
        // error: dimension mismatch
        error("int_appendcols: dimension mismatch");
        }
          }
          return(NULL);
        }
/* transpose */
int_matrix *int_transp(int_matrix *m)
{
  int_matrix *c;
  int i, j;
  c = NULL;
  if (m != NULL)
    {
      if ((c = int_copy_temp(m)) != NULL)
        {
          c->rows = m->cols;
          c->cols = m->rows;
          for (j = 0; j < m->cols; j++)
            for (i = 0; i < m->rows; i++)
              c->re[j + i * c->rows] = m->re[i + j * m->rows];
          if (m->type == CMPLX)
            {
              for (j = 0; j < m->cols; j++)
                for (i = 0; i < m->rows; i++)
                  c->im[j + i * c->rows] = m->im[i + j * m->rows];
            }
        }
    }
  return(c);
}
int_matrix *int_herm(int_matrix *m)
{
  int_matrix *c;
```

```
        int i, j;
        c = NULL;
        if (m != NULL)
          {
            if ((c = int_copy_temp(m)) != NULL)
              {
                c->rows = m->cols;
                c->cols = m->rows;
                for (j = 0; j < m->cols; j++)
                  for (i = 0; i < m->rows; i++)
                    c->re[j + i * c->rows] = m->re[i + j * m->rows];
                if (m->type == CMPLX)
                  {
                    for (j = 0; j < m->cols; j++)
                      for (i = 0; i < m->rows; i++)
                        c->im[j + i * c->rows] = -m->im[i + j * m->rows];
                  }
              }
          }
        return(c);
}
/* return all cols of m and rows indexed by a
   Matlab:
   >> m(a,:) % return all cols and rows of m indexed by a
*/
int_matrix *int_index_rows(int_matrix *m, int_matrix *a) /* All cols, some rows */
     /* int_matrix *m   source matrix */
     /* int_matrix *a   index matrix */
{
  int_matrix *c;
  int i, j, k;
  INTTYPE *ci, *ai, *mi;
  if (m != NULL && a != NULL)
    {
      a = int_real(a);                     /* real integer indicies only */
      if (a->re[int_min_index(a)] > 0
          && a->re[int_max_index(a)] <= m->rows) /* are indicies in range */
        {
          if ((c = int_new_temp(SIZE(a), m->cols, m->type)) != NULL)
            {
              k = 0;
              ci = c->re;
              mi = m->re;
              ai = a->re;        /* index real part */
              for (j = 0; j < m->cols; j++) /* for all cols in m */
                for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                  ci[k++] = mi[ai[i] - 1 + j * m->cols]; /* get column */
              if (m->type == CMPLX) /* index imaginary part if needed */
                {
                  k = 0;
                  ci = c->im;
                  mi = m->im;
                  ai = a->re;
                  for (j = 0; j < m->cols; j++) /* for all cols in m */
                    for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                      ci[k++] = mi[ai[i] - 1 + j * m->cols]; /* get col */
                }
            }
          return(c);                /* done */
        }
    }
  return(NULL);
}
/* return all rows of m and cols indexed by a
```

```
    Matlab:
    >> m(:,a) % return all rows and cols of m indexed by a
    >> m(:)    % return m as a column vector (a == NULL)
*/
int_matrix *int_index_cols(int_matrix *m, int_matrix *a) /* All rows, some cols */
    /* int_matrix *m:  source matrix */
    /* int_matrix *a:  index matrix */
{
  int_matrix *c;
  int i, j, k;
  INTTYPE *ci, *mi, *ai;
  if (m != NULL)                 /* if source matrix is not NULL */
    {
      if (a != NULL)             /* if index matrix is not NULL */
        {
          a = int_real(a);           /* real indicies only */
          if (a->re[int_min_index(a)] > 0
              && a->re[int_max_index(a)] <= m->rows) /* are indicies in range */
            {
              if ((c = int_new_temp(m->rows, SIZE(a), m->type)) != NULL)
                {
                  k = 0;
                  ci = c->re;
                  mi = m->re;
                  ai = a->re;   /* index the real part */
                  for (j = 0; j < m->cols; j++) /* for the whole row of m */
                    for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                      ci[k++] = mi[j + (ai[i] - 1) * m->cols];
                  if (m->type == CMPLX) /* index imaginary part if needed */
                    {
                      k = 0;
                      ci = c->im;
                      mi = m->im;
                      for (j = 0; j < m->cols; j++) /* for whole row of m */
                        for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                          ci[k++] = mi[j + (ai[i] - 1) * m->cols];
                    }
                }
              return(c);           /* done */
            }
          else
            {
              // error: indices out of range
              error("int_index_cols: indices out of range");
            }
        }
      else
        {
            c = int_copy_temp(m);  /* if a == NULL, make m a row vector, */
            c->rows = SIZE(m); /* this is a matlab convention        */
            c->cols = 1;
            return(c);           /* done */
        }
    }
  return(NULL);
}
/* return m row-indexed by a and column-indexed by b
    Matlab:
    >> m(a,b) % indexes both rows and cols
    >> m(a)   % indexes m as column vector, but returns a matrix with a's
              % dimensions (b == NULL)
*/
int_matrix *int_index_rows_cols(int_matrix *m, int_matrix *a, int_matrix *b)
    /* int_matrix *m:  source matrix */
    /* int_matrix *a:  row index matrix */
```

```
        /* int_matrix *b;  column index matrix */
{
int_matrix *c;
int i, j, k;
INTTYPE *ci, *ai, *bi, *mi;
if (m != NULL && a != NULL)
    {
        a = int_real(a);                /* only real indicies */
        b = int_real(b);
        if (b == NULL)              /* index m as a column vector
                                       and use a's dimensions */
          {
            if (a->re[int_min_index(a)] > 0
                && a->re[int_max_index(a)] <= SIZE(m)) /* indicies in range */
               {
                if ((c = int_copy_temp(a)) != NULL) /* copy a */
                   {
                    ci = c->re;
                    mi = m->re;
                    ai = a->re;
                    for (i = 0; i < SIZE(a); i++) /* index m */
                      ci[i] = mi[ai[i] - 1];
                    if (m->type == CMPLX) /* index imaginary part if needed */
                       {
                        ci = c->im;
                        mi = m->im;
                        for (i = 0; i < SIZE(a); i++) /* index m */
                          ci[i] = mi[ai[i] - 1];
                       }
                   }
                return(c);          /* done */
               }
            else
               {
                // error: indicies out of range
                error("int_index_rows_cols: indices out of range");
               }
          }
        if (b != NULL)              /* index both rows and cols of m */
          {
            if (a->re[int_min_index(a)] > 0 && b->re[int_min_index(b)] > 0 &&
                a->re[int_max_index(a)] <= m->rows && b->re[int_max_index(b)] <= m->cols)
               {
                if ((c = int_new_temp(SIZE(a), SIZE(b), m->type)) != NULL)
                   {
                    ci = c->re;
                    mi = m->re;
                    ai = a->re;
                    bi = b->re;
                    k = 0;          /* index real part */
                    for (j = 0; j < SIZE(b); j++) /* for all cols ind. */
                      for (i = 0; i < SIZE(a); i++) /* for all rows ind. */
                        ci[k++] = mi[ai[i]-1 + (bi[j] - 1) * m->cols];
                    if (m->type == CMPLX) /* index imaginary part if needed */
                       {
                        ci = c->im;
                        mi = m->im;
                        k = 0;
                        for (j = 0; j < SIZE(b); j++) /* for all cols ind. */
                          for (i = 0; i < SIZE(a); i++) /* for all row ind. */
                            ci[k++] = mi[ai[i] - 1 +
                                       (bi[j] - 1) * m->cols];
                       }
                   }
                return(c);          /* done */
```

```
            }
          else
          {
          // error: indices out of range
           error("int_index_rows_cols: indices out of range");
          }
        }
      }
    return(NULL);
}
/* make matrix of a from:step:to statement,
   Matlab:
   >> from:step:to  %
   >> from:to       % step == NULL -> step size = 1
*/
int_matrix *int_range(INTTYPE from, INTTYPE step, INTTYPE to)
{
  int_matrix *m;
  int i;
  INTTYPE j;
  m = NULL;
  if (step == 0)          /* if step == 0, set step size = 1 */
    step = 1;
if (to < from && step < 0)      /* if to < from then step must be < 0 */
{
if ((m = int_new_temp(1, INT(1 + (from - to) / -step), REAL)) != NULL)
{
j = from;          /* start at from */
for (i = 0; i < SIZE(m); i++) /* for all elem. in m */
          {
            m->re[i] = j;
            j += step;       /* update by step size */
          }
      }
  }
else                      /* step size > 0 and to > from */
{
if ((m = int_new_temp(1, INT(1 + (to - from) / step), REAL)) != NULL)
{
j = from;
        for (i = 0; i < SIZE(m); i++)
          {
            m->re[i] = j;
            j += step;       /* update with step size */
          }
      }
    }
  return(m);                    /* done */
}
/* Returns a copy of mat according to the MATLAB expression
   >>m(row, col)
   For ':' use NULL for row or col, that is
   >>m(row,:)
   would be coded as submatrix(mat, row, NULL)  */
int_matrix *int_sub_matrix(int_matrix *mat, int_matrix *rows, int_matrix *cols)
{
if (mat != NULL)
{
if (rows != NULL && cols != NULL)
return(int_index_rows_cols(mat, rows, cols)); /* mat(rows, cols) */
if (rows == NULL && cols != NULL) /* mat(:, cols) */
    return(int_index_cols(mat, cols));
if (rows != NULL && cols    NULL) /* mat(rows, :) */
    return(int_index_rows(mat, rows));
```

```
return(int_copy_temp(mat)); /* mat(:,:) */
}
return(mat);
}
/* Assigns source to a submatrix of target indexed by rows and cols.
>>target(rows, cols) = source
rows and cols must be vectors or NULL, NULL is interpreted as :, that is
>>target(rows,:) = source
would be coded as assign(target, rows, NULL, source).
NOTE: does not handle the case
>>target(:,:) = source
use target = int_copy_temp(source), for this case
*/
int_matrix *int_assign(int_matrix *target, int_matrix *rows, int_matrix *cols, int_matrix *sou
{
INTTYPE *r, *c;
int i, j, sr, tr;
if (target != NULL)
{
if (rows != NULL && cols != NULL)  /* target(rows, cols) = source */
{
rows = int_real(rows);  /* Only real integer indicies */
cols = int_real(cols);  /* Only real integer indicies */
                                        /* make sure dimensions match */
if (rows->re[int_max_index(rows)] <= target->rows     /* max(rows) <= # of rows in target */
    && cols->re[int_max_index(cols)] <= target->cols /* max(cols) <= # of cols in target */
&& rows->re[int_min_index(rows)] > 0  /* min(rows) > 0 */
&& cols->re[int_min_index(cols)] > 0   /* min(cols) > 0 */
&& SIZE(rows) * SIZE(cols) == SIZE(source) )  /* # of elements indexed in target = # of elemer
{
int_cmplx_promote(target, source);
r = rows->re;
c = cols->re;
sr = source->rows;
tr = target->rows;
for (i = 0; i < SIZE(rows); i++)
for (j = 0; j < SIZE(cols); j++)
{
target->re[r[i]-1 + c[j]-1 * tr] = source->re[i + j * sr];
if (target->type == CMPLX);
target->im[r[i]-1 + c[j]-1 * tr] = source->im[i + j * sr];
}
}
else
{
// error: dimension mismatch
error("int_assign: dimension mismatch");
}
}
if (rows != NULL && cols == NULL)  /* target(rows, :) = source */
{
rows = int_real(rows);  /* Only real integer indicies */
                                        /* make sure dimensions match */
if (rows->re[int_max_index(rows)] <= target->rows     /* max(rows) <= # of rows in target */
&& rows->re[int_min_index(rows)] > 0  /* min(rows) > 0 */
&& target->cols == source->cols)               /* target cols = source cols */
{
int_cmplx_promote(target, source);
r = rows->re;
sr = source->rows;
tr = target->rows;
for (i = 0; i < SIZE(rows); i++)
for (j = 0; j < target->cols; j++)
```

```
{
target->re[r[i]-1 + j + tr] = source->re[i + j * sr];
if (target->type == CMPLX);
target->im[r[i]-1 + j + tr] = source->im[i + j * sr];
}
}
else
{
// error: dimension mismatch
error("int_assign: dimension mismatch");
}
}
if (rows == NULL && cols != NULL)  /* target(:, cols) = source */
{
cols = int_real(cols);  /* Only real integer indicies */
                                  /* make sure dimensions match */
if (cols->re[int_max_index(cols)] <= target->cols     /* max(cols) <= # of cols in target */
&& cols->re[int_min_index(cols)] > 0  /* min(cols) > 0 */
&& target->rows == source->rows)              /* target rows = source rows */
{
int_cmplx_promote(target, source);
c = cols->re;
sr = source->rows;
tr = target->rows;
for (i = 0; i < target->rows; i++)
for (j = 0; j < SIZE(cols); j++)
{
target->re[i + c[j]-1 * tr] = source->re[i + j * sr];
if (target->type == CMPLX);
target->im[i + c[j]-1 + tr] = source->im[i + j * sr];
}
}
else
{
// error: dimension mismatch
error("int_assign: dimension mismatch");
}
}
return(target);
}
return(NULL);
}
/* Create a 1x1 matrix from a scalar */
int_matrix *int_scl2mat(INTTYPE re, INTTYPE im, int type)
{
  int_matrix *tmp;
  if ((tmp = int_new_matrix(1, 1, type)) != NULL)
    {
      tmp->re[0] = re;
      if (type == CMPLX)
          tmp->im[0] = im;
    }
  return(tmp);
}
```

II.6 LIST.H

```
/* Header file for makelist */
/* Usage: e.g. deref(int,x) or deref(char *,x) */
#define deref(type,x)  *((type*)(x))
```

```
typedef struct data DATA;
typedef struct list LIST;
typedef struct prop PROP;
struct data {
  void *data;    /* space for list->nentries instances of data */
  DATA *next;    /* next list->nentries collection of data */
};
struct prop {
  void *dataptr;          /* to what data item this property associates */
  void *propval;          /* property value to associate with the data */
  void *propsym;          /* optional symbol (usually char *) to associate */
  PROP *next;
};
struct list {
  int  entrysize;                  /* size of each data entry in bytes */
  int  nentries;                   /* # entries to grab per malloc call */
  int  empty_slots;     /* empty slots left in current data block */
  int  nitems;          /* total items saved in this list */
  int  ecount;          /* where we are when reading back list */
  int  fblock;
  DATA *fdata;
  PROP *prop;           /* optional property list for this list */
  DATA *data;           /* linked list for the actual data of this list */
  DATA *hidata;         /* highest allocated data block (for efficiency) */
};
/* Internal malloc routine */
/*#define MEMCHK*/
#ifdef MEMCHK
static FILE *memfp = NULL;
#endif
/* Function prototypes/declarations */
LIST *makelist(int esize, int nentries);
void *putproplist(LIST *list, void *dataptr, char *propsym, void *val);
LIST *getproplist(LIST *list, void *dataptr, char *propsym);
LIST *findprop(LIST *list, char *propsym);
LIST *poplist(LIST *list);
void *toplist(LIST *list);
void *appendlist(LIST *list, void *data);
int listindex(LIST *list);
void *pappendlist(LIST *list, void *data, char *propsym, void *val);
void *pushlist(LIST *list, void *data);
void *fetchlist(LIST *list, int ndx);
void rewindlist(LIST *list);
void *walklist(LIST *list);
void *malloclist(LIST *list, int size);
int freelist(LIST *list);
int gclist(LIST *list);
```

II.7 LIST.C

```
/*      makelist- list management package
        RF Starr
        2639 Valley Field Dr.
        SugarLand, TX 77479
        */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include "list.h"
```

```
/*#define DEBUG*/
#ifdef DEBUG
#define Debug(x) x
#else
#define Debug(x)
#endif
static void *imalloc(int size)
{
  void *ptr = malloc(size);
#ifdef MEMCHK
  if (!memfp) memfp = fopen("meminfo","w");
#endif
  if (!ptr)
    {
      fprintf(stderr,"malloc error: no free memory left.\n");
      fflush(stderr);
    }
#ifdef MEMCHK
  fprintf(memfp,"%x malloc\n",ptr);
  fflush(memfp);
#endif
  return ptr;
}
static void *ifree(void *addr)
{
  free(addr);
#ifdef MEMCHK
  fprintf(memfp,"%x free\n",addr);
  fflush(memfp);
#endif
}
/* Build, initialize, and return an empty list */
LIST *makelist(int esize, int nentries)
{
  LIST *list = (LIST*)imalloc(sizeof(LIST)+sizeof(DATA)+esize*nentries);
  void *dp   = imalloc(sizeof(DATA)+esize*nentries);
  if (!list || !dp) return NULL;
  list->data = (DATA *)dp;
  list->data->data = (char *)dp + sizeof(DATA);
  list->entrysize = esize;
  list->nentries = nentries;
  list->empty_slots = nentries;
  list->nitems = 0;
  list->ecount = 0;
  list->fblock = 0;
  list->fdata = NULL;
  list->prop = (PROP *)NULL;
  list->hidata = list->data;
  list->data->next = NULL;
  return (void *)list;
}
/*      Put items on property list for this data item.  Propsym is the
        property symbol, and val is a pointer to a _static_ are where the
        data for this property resides.
        */
void *putproplist(LIST *list, void *dataptr, char *propsym, void *val)
{
  PROP *newprop = (PROP *)imalloc(sizeof(PROP));
  PROP *topprop = list->prop;
  if (!list) return NULL;
  if (!newprop) return NULL;
  newprop->dataptr = dataptr;
  newprop->propsym = propsym;
  newprop->propval = val;
```

282

```
      newprop->next = topprop;
      list->prop = newprop;
}
/*      Read an item off of the property list for a particular data
        item.  NULL returned if there is none.
        */
LIST *getproplist(LIST *list, void *dataptr, char *propsym)
{
  PROP *p;
  void *propval = NULL;
  if (!list) return NULL;
  p = list->prop;
  while (p) {
    int  fsym = !strcmp(p->propsym,propsym);
    if ((!dataptr && fsym) || (p->dataptr == dataptr && fsym)) {
      propval = p->propval;
      break;
    }
    p = p->next;
  }
  return propval;
}
    /* Find data item associated with a property name */
LIST *findprop(LIST *list, char *propsym)
{
  PROP *p;
  if (!list) return NULL;
  p = list->prop;
  while (p) {
    if (!strcmp(p->propsym,propsym)) return p->dataptr;
    p = p->next;
  }
  return NULL;
}
/* Append data to the specified list */
static void  whereis(LIST *list, int ndx, int *walk, int *put)
{
  *walk = ndx / list->nentries;
  *put  = ndx % list->nentries;
}
/* Remove last entry on the specified list... adjust struct accordingly */
LIST *poplist(LIST *list)
{
  DATA *org;
  void *dp = NULL;
  unsigned char *data;
  int  put;
  if (!list) return NULL;
  if (!list->nitems) return dp;
  org = list->data;
  list->empty_slots++;
  list->nitems--;
  if (list->empty_slots == list->nentries)
    {
      while (org->next) org = org->next;
      put  = list->nitems % list->nentries;
      data = (unsigned char *)org->data;
      dp = (void *)(data+(list->entrysize*put));
      if (org->next) ifree(org->next), org->next = NULL;
      list->hidata = org;
    }
  return dp;
```

```
}
/* calculate data pointer for the ndx entry */
static void *calcdp(LIST *list, int ndx)
{
  DATA *pdata;
  unsigned char *dp;
  int  size = (list) ? list->entrysize :0;
  int  max  = (list) ? list->nitems : 0;
  int  walk,put;
  if (!list || ndx >= max) return (void *)NULL;
  whereis(list,ndx,&walk,&put);
  pdata = list->data;
  while (walk--) pdata = pdata->next;
  dp = (unsigned char *)pdata->data;
  return (void *)(dp+(size*put));
}
/* Return pointer to data which is last on the list */
void *toplist(LIST *list)
{
  void *dp = NULL;
  unsigned char *data;
  int put;
  if (!list || !list->nitems) return NULL;
  return calcdp(list,list->nitems-1);
}
/* append a data item onto specified list */
void *appendlist(LIST *list, void *data)
{
  DATA *pdata;
  void *where;
  unsigned char *dp;
  int  walk,put,size;
  if (!list) return NULL; // error: invalid list
  size = list->entrysize;
  whereis(list,list->nitems,&walk,&put);
  pdata = list->hidata;
  if (!list->empty_slots)
    {
      void *mem = imalloc(sizeof(DATA)+size*list->nentries);
      if (!mem) return NULL;
      list->hidata = pdata = pdata->next = (DATA *)mem;
      pdata->data = (void *)((char *)mem+sizeof(DATA));
      pdata->next = NULL;
      list->empty_slots = list->nentries;
    }
  dp = (unsigned char *)pdata->data;
  where = (char *)(dp+(put*size));
  memcpy(where,data,size);
  list->empty_slots--;
  list->nitems++;
  return where;
}
/* return index of NEXT list entry */
int listindex(LIST *list)
{
  return list->nitems;
}
/* append a data item onto specified list with property value information */
void *pappendlist(LIST *list, void *data, char *propsym, void *val)
{
  void *dataptr = appendlist(list,data);
  if (dataptr) putproplist(list,dataptr,propsym,val);
```

```
    return dataptr;
}
/* Just like appendlist... new name for compatibility with poplist */
void *pushlist(LIST *list, void *data)
{
  if (!list) return NULL;
  return appendlist(list,data);
}
/* Fetch a specific data item off of list... ndx is 0-based */
void *fetchlist(LIST *list, int ndx)
{
  DATA *pdata;
  unsigned char *dp;
  int  size = list->entrysize;
  int  max  = list->nitems;
  int  walk,put;
  if (!list) return NULL;
  if (ndx >= max) return (void *)NULL;
  whereis(list,ndx,&walk,&put);
  if (walk == list->fblock)
    list->fdata = pdata = (list->fdata) ? list->fdata : list->data;
  else
    {
      pdata = list->data;
      list->fblock = walk;
      while (walk--)
        pdata = pdata->next;
      list->fdata = pdata;
    }
  dp = (unsigned char *)pdata->data;
  Debug(printf("fetchlist: getting data from %x\n",dp+(size*put)));
  return (char *)(dp+(size*put));
}
/* reset pointer used by walklist */
void rewindlist(LIST *list)
{
  if (list) list->ecount = 0;
}
/* walk down the list, returning each data item 'till there ain't no more */
void *walklist(LIST *list)
{
  DATA *pdata;
  unsigned char *dp;
  int  size = list->entrysize;
  int  max  = list->nitems;
  int  index = list->ecount;
  int  walk,put;
  if (!list) return NULL;
  dp = (unsigned char *)fetchlist(list,index);
  if (!dp)
    list->ecount = 0;
  else
    list->ecount++;
  return (list->ecount) ? (void *)dp : (void *)NULL;
}
/* malloc size bytes, and add address of malloced space to the list */
void *malloclist(LIST *list, int size)
{
  void *dp;
  if (!list) return NULL;
  dp = imalloc(size);
  if (dp) appendlist(list,&dp);
  return dp;
```

```
}
/* Free up all malloced data associated with the specified list */
int freelist(LIST *list)
{
  fl(list,0);
}
/* garbage collect list... assume all data in list is malloced ptrs */
int gclist(LIST *list)
{
  fl(list,1);
}
/* Free the list up.  If freedp != 0, free each data pointer as well */
static int fl(LIST *list, int freedp)
{
  DATA *pdata,*ppd;
  PROP *p = list->prop;
  if (!list) return;
  pdata = list->data;
  if (freedp)
    {
      void *dp;
      rewindlist(list);
      while (dp=walklist(list))
        ifree(*(char **)dp);
    }
  /* free all malloced data */
  while (pdata)
    {
      DATA *next = pdata->next;
      ifree(pdata);
      pdata = next;
    }
  /* free all malloced property info */
  while (p)
    {
      PROP *n = p->next;
      ifree(p);
      p = n;
    }
  ifree(list);
}
/*      given a list of functions, invoke the function with the specified
        property tag with the arguments supplied.
        */
```

H.8   MATRIX.H

```
/*********** matrix.h ********************/
 /* Gauss Machine High-Level API  */

#ifndef __matrix_h /* Include only once */
#define __matrix_h
/* Calulates the number if elements in matrix a */
#define SIZE(a) ((a)->rows * (a)->cols)
/* Are a and b of the same dimensions? */
#define EQDIM(a, b) ( (a->rows == b->rows) &&  (a->cols == b->cols) )
/* Complex multiply. (a + jb) = (c + jd) * (d + je) */
#define cmul(a, b, c, d, e, f); {  a = (c) * (e) - (d) * (f);  b = (c) * (f) + (d) * (e); }
/* Complex magnitude */
#define cabs(a, b) sqrt(((a) * (a) + (b) * (b)))
```

```
/*  int cmplx_promote(matrix *a, matrix *b)
description:
Promotes, if necessary, the operands a and b to complex matrices, that is,
if either a or b is complex then both a and b are converted to complex matrices.
arguments:
matrix *a, *b = matrices to be promoted
returns:
int =  0  if successful
-1 if malloc failed
usage:
res = cmplx_promote(a, b); // complex promotes a and b, OK if res == 0
see also:
op_check()
*/
int cmplx_promote(matrix *a, matrix *b);

/* int op_check(matrix *a, matrix *b)
description:
Check field type and dimensions. First a and b are complex promoted by
cmplx_promote(). Returns COMP if a and b have the same dimensions.
Returns COMP | SCAL if either a or b is a scalar. In addition, if a and b
are complex, CMPLX is ORed to the returned value otherwise CMPLX is ORed to
the returned value. Thus, if type = op_check(a,b), then (type & CMPLX) will
be true if either a or b is complex, (type & REAL) will be true if both a and
b are real, (type & COMP) will be true if dimensions match and if a or be is
a scalar, (type & SCAL) will be true if a or b is a scalar.
arguments:
matrix *a, *b = matrices to be checked
returns:
int = whose bits are set according to the description above.
usage:
type = op_check(a, b); // checks dimensions of a, b. Bits will be set in type
// according to the description above
see also:
cmplx_promote(), mul_check()
note:
    op_check() is used by add(), pmul(), pdiv()
*/
int op_check(matrix *a, matrix *b);

/* int mul_check(matrix *a, matrix *b)
description:
Check field type and dimensions. First a and b are complex promoted by
cmplx_promote(). Returns COMP if a and b have the same dimensions.
Returns COMP | SCAL if either a or b is a scalar. In addition, if a and b
are complex, CMPLX is ORed to the returned value otherwise CMPLX is ORed to
the returned value. Thus, if type = op_check(a,b), then (type & CMPLX) will
be true if either a or b is complex, (type & REAL) will be true if both a and
b are real, (type & COMP) will be true if dimensions match and if a or be is
a scalar, (type & SCAL) will be true if a or b is a scalar.
arguments:
matrix *a, *b  matrices to be checked
returns:
int whose bits are set according to the description above.
usage:
type = mul_check(a, b); // checks dimensions of a, b. Bits will be set in type
// according to the description above
see also:
cmplx_promote(), op_check()
note:
    mul_check() is used by mul()
*/
int mul_check(matrix *a, matrix *b);
/* matrix *scl2mat(FLOATTYPE re, FLOATTYPE im, int type)
description:
Create a 1x1 matrix from a scalar
arguments:
FLOATTYPE re, im real part and imaginary part of matrix
```

```
int type REAL if real or CMPLX if complex
returns:
matrix * 1x1 matrix with re as real part and im as imaginary part (if complex)
usage:
m = scl2mat(4.7, 1.1, CMPLX); // m is a 1x1 complex matrix, m(1) = 4.7 + j1.1
m = scl2mat(4.7, 1.1, REAL);  // m is a 1x1 real matrix, m(1) = 4.7
*/
matrix *scl2mat(FLOATTYPE re, FLOATTYPE im, int type);
/****************** Memory management of matrices ********************/
/* matrix *new_matrix(int rows, int cols, int type)
description:
Allocates memory for a matrix with dimension rows and cols and of type type.
Note that it is the user's responsibility to free this matrix. Use new_temp()
to get a temporary matrix (which will be freed by kill_temp_list()
or close_GM()).
arguments:
int rows, cols  dimensions of matrix to be allocated
int type type of matrix, CMPLX for a complex matrix and REAL for a
        real matrix
returns:
matrix *  matrix of the requested size and type. NULL if out of memory.
usage:
mat = new_matrix(3, 5, CMPLX);  // Allocate memory for a 3x5 complex matrix
see also:
copy_matrix(), new_temp(), copy_temp(), kill_matrix(), kill_temp_list()
note:
It is the user's responsibility to free any matrix that has been allocated with
new_matrix (with kill_matrix).
*/
matrix *new_matrix(int rows, int cols, int type);
/* void kill_matrix(matrix *m)
description:
Frees memory used by m. Note that m should have been allocated with new_matrix()
or copy_matrix().
If m is a temporary matrix, that is, allocated explicitly or implicitly with
new_temp() or copy_temp, then kill_temp_list() should be used instead of
kill_matrix().
arguments:
matrix *m Matrix to be freed, must have been allocated with new_matrix()
or copy_matrix()
returns:
nothing
usage:
kill_matrix(A);  // Free memory allocated for A
see also:
copy_matrix(), new_temp(), copy_temp(), new_matrix(), kill_temp_list()
note:
m must have been allocated explicitly or implicitly with new_matrix()
or copy_matrix()
*/
void kill_matrix(matrix *m);
/* matrix *copy_matrix(matrix *source)
description:
Returns a copy of the matrix source. The copy is allocated with new_matrix().
Note that it is the user's responsibility to free this matrix. Use copy_temp()
to get a temporary matrix (which will be freed by kill_temp_list()
or close_GM()).
arguments:
matrix *source  matrix to be copied
returns:
matrix *  copy of source. NULL if out of memory.
usage:
new = copy_matrix(old);  // copy the matrix old to the matrix new
see also:
kill_matrix(), new_temp(), copy_temp(), new_matrix(), kill_temp_list()
note:
```

It is the user's responsibility to free any matrix that has been allocated with
copy_matrix (with kill_matrix).
*/
matrix *copy_matrix(matrix *source);
/* matrix *new_temp(int rows, int cols, int type)
description:
Allocates memory for a temporary matrix with dimension rows and cols and
of type type.
Note that it is the user's responsibility to free this matrix. To free ALL
temporary matrices, use kill_temp_list() or close_GM().
arguments:
int rows, cols dimensions of matrix to be allocated
int type type of matrix, CMPLX for a complex matrix and REAL for a
        real matrix
returns:
matrix *   matrix of the requested size and type. NULL if out of memory.
usage:
mat = new_temp(3, 5, CMPLX);// Allocate memory for a temporary 3x5 complex matrix
see also:
kill_matrix(), copy_matrix(), copy_temp(), new_matrix(), kill_temp_list()
*/
matrix *new_temp(int rows, int cols, int type);
/* matrix *copy_temp(matrix *source)
description:
Returns a copy of the matrix source. The copy is allocated with new_temp()
and therfore, is a temporary matrix.
To free ALL temporary matrices, use kill_temp_list() or close_GM().
arguments:
matrix *source matrix to be copied
returns:
matrix *   copy of source. NULL if out of memory.
usage:
new = copy_temp(old);   // copy the matrix old to the temporary matrix new
see also:
kill_matrix(), copy_matrix(), new_temp(), new_matrix(), kill_temp_list()
*/
matrix *copy_temp(matrix *source);
/* void kill_temp_list(void)
description:
Frees memory allocated by ALL temporary matrices. The temporary matrices are
allocated, explicitly or implicitly, with new_temp() or copy_temp.
arguments:
none
returns:
nothing
usage:
kill_temp_list(); // Free memory allocated by all temporary matrices
see also:
kill_matrix(), copy_matrix(), new_temp(), new_matrix(), copy_temp(), close_GM()
*/
void kill_temp_list(void);
/* init_temp_list(void)
description:
Initialize the list for allocation of temporary matrices
arguments:
none
returns:
int NOERR if alright, -1 if out of memory
usage:
err = init_temp_list(); // Initialize temp matrices, inspect err for errors.
see also:
init_GM()
note:
init_temp_list() is normally called from initGM();
*/
int init_temp_list(void);

```
/*  matrix *real(matrix *mat)
description:
Returns a matrix containing the real part of mat
arguments:
matrix *mat   input matrix
returns:
matrix * real part of mat
usage:
real_part = real(cmplx_matrix); // real_part = Re[cmplx_matrix]
MATLAB equivalent:
>> real_part = real(cmplx_matrix);
*/
matrix *real(matrix *mat);
/*  matrix *imag(matrix *mat)
description:
Returns a matrix containing the imaginary part of mat
arguments:
matrix *mat   input matrix
returns:
matrix * imaginary part of mat
usage:
im_part = imag(cmplx_matrix); // im_part = Im[cmplx_matrix]
MATLAB equivalent:
>> im_part = imag(cmplx_matrix);
*/
matrix *imag(matrix *mat);
/*  int max_index(matrix *mat)
description:
Return index to maximal element of mat
arguments:
matrix *mat   input matrix
returns:
int index to the maximal element of mat
usage:
max_i = max_index(mat); // max(mat) = mat[max_i]
MATLAB equivalent:
>> max_i = find(mat == max(mat(:)));
*/
int max_index(matrix *mat);
/*  int min_index(matrix *mat)
description:
Return index to minimal element of mat
arguments:
matrix *mat   input matrix
returns:
int index to the minimal element of mat
usage:
min_i = min_index(mat); // min(mat) = mat[min_i]
MATLAB equivalent:
>> min_i = find(mat == min(mat(:)));
*/
int min_index(matrix *mat);
/*  matrix *minus(matrix *mat)
description:
Negates elements of mat
arguments:
matrix *mat   input matrix
returns:
matrix * the negated input matrix
usage:
minus_A = minus(A); // minus_A = -A
MATLAB equivalent:
>> minus_A = -A;
*/
matrix *minus(matrix *mat);
/*  matrix *conj(matrix *mat)
description:
```

```
conjugates elements of mat
arguments:
matrix *mat   input matrix
returns:
matrix * the conjugated input matrix
usage:
conj_A = conj(A); // conj_A = -A
MATLAB equivalent:
>> conj_A = conj(A);
*/
matrix *conj(matrix *mat);
/*  matrix *add(matrix *a, matrix *b)
description:
adds matrices a and b
arguments:
matrix *a, b   input matrices
returns:
matrix * the sum of a and b, NULL if error
usage:
sum = add(a, b); // sum = a + b
MATLAB equivalent:
>> sum = a + b;
*/
matrix *add(matrix *a, matrix *b);
/*  matrix *pmul(matrix *a, matrix *b)
description:
pointwise multiplication of matrices a and b
arguments:
matrix *a, b   input matrices
returns:
matrix * the pointwise product of a and b, NULL if error
usage:
pprod = pmul(a, b); // pprod = a .* b
MATLAB equivalent:
>> pprod = a .* b;
*/
matrix *pmul(matrix *a, matrix *b);
/*  matrix *pdiv(matrix *a, matrix *b)
description:
pointwise division of matrices a and b
arguments:
matrix *a, b   input matrices
returns:
matrix * the pointwise division of a and b, NULL if error
usage:
pprod = pdiv(a, b); // pdiv = a ./ b
MATLAB equivalent:
>> pprod = a ./ b;
*/
matrix *pdiv(matrix *a, matrix *b);
/*  matrix *mul(matrix *a, matrix *b)
description:
multiplication of matrices a and b
arguments:
matrix *a, b   input matrices
returns:
matrix * the product of a and b, NULL if error
usage:
prod = pmul(a, b); // pprod = a * b
MATLAB equivalent:
>> prod = a * b;
*/
matrix *mul(matrix *a, matrix *b);
/*  matrix *appendrows(matrix *a, matrix *b)
description:
appends b's rows to a
```

```
arguments:
matrix *a, b   input matrices
returns:
matrix * [a; b], NULL if error
usage:
c = appendrows(a, b); // c = [a; b]
MATLAB equivalent:
>> c = [a; b];
*/
matrix *appendrows(matrix *a, matrix *b);
/*  matrix *appendcols(matrix *a, matrix *b)
description:
appends b's columns to a
arguments:
matrix *a, b   input matrices
returns:
matrix * [a, b], NULL if error
usage:
c = appendcols(a, b); // c = [a, b]
MATLAB equivalent:
>> c = [a, b];
*/
matrix *appendcols(matrix *a, matrix *b);
/*  matrix *transp(matrix *mat)
description:
transposes elements of mat. Note: does not conjugate elements. Use herm()
for conjugate transpose (hermitian).
arguments:
matrix *mat   input matrix
returns:
matrix * the transposed input matrix, NULL if error.
usage:
tran_A = transp(A); // tran_A = A'
MATLAB equivalent:
>> A = A.'; % Note: ".'" not "'", that is, does not conjugate transpose
see also:
herm()
*/
matrix *transp(matrix *mat);
/*  matrix *herm(matrix *mat)
description:
conjugate transposes elements of mat, that is, takes the hermitian of mat.
arguments:
matrix *mat   input matrix
returns:
matrix * the conjugate transposed input matrix, NULL if error.
usage:
tran_A = transp(A); // tran_A = A'*
MATLAB equivalent:
>> A = A'; % Note: "'" not ".'", that is, conjugate transpose
see also:
transp()
*/
matrix *herm(matrix *mat);
/*  matrix *mfloor(matrix *mat)
description:
truncates elements of mat to closest smaller integer value, that is, floor function.
arguments:
matrix *mat   input matrix
returns:
matrix * the "floored" input matrix, NULL if error.
usage:
int_A = mfloor(A); // int_A = A'*
MATLAB equivalent:
>> int_A = floor(A);
note:
```

```
the output is still a floating point matrix, even though the elements are
truncated to integers
*/
matrix *mfloor(matrix *mat);
/*  matrix *index_rows(matrix *mat, matrix *ind)
description:
Returns the rows of mat that are pointed out by ind. The elements of ind are
truncated to integers (with mfloor) and are used to pick out rows. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3.14 1.99], then the result
would be a matrix of the form [7 8 9; 4 5 6]. This is analoguous to the MATLAB
statement, mat(ind, :).
arguments:
matrix *mat   matrix to be indexed
matrix *ind   row indexing matrix
returns:
matrix * the indexed input matrix, NULL if error.
usage:
B = index_rows(mat, ind); // B = mat(ind, :)
MATLAB equivalent:
>> B = mat(ind, :)
see also:
index_cols(), index_rows_cols(), sub_matrix()
note:
For greatest convenience, use sub_matrix() for all indexing purposes.
*/
matrix *index_rows(matrix *mat, matrix *ind);
/*  matrix *index_cols(matrix *mat, matrix *ind)
description:
Returns the columns of mat that are pointed out by ind. The elements of ind are
truncated to integers (with mfloor) and are used to pick out columns. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3.14 1.99], then the result
would be a matrix of the form [3 2; 6 5; 9 8]. This is analoguous to the MATLAB
statement, mat(:, ind).
arguments:
matrix *mat   matrix to be indexed
matrix *ind   column indexing matrix
returns:
matrix * the indexed input matrix, NULL if error.
usage:
B = index_cols(mat, ind); // B = mat(:, ind)
MATLAB equivalent:
>> B = mat(:, ind)
see also:
index_rows(), index_rows_cols(), sub_matrix()
note:
For greatest convenience, use sub_matrix() for all indexing purposes.
*/
matrix *index_cols(matrix *mat, matrix *ind);
/*  matrix *index_rows_cols(matrix *mat, matrix *ind_a, matrix *ind_b)
description:
Returns the rows and columns of mat that are pointed out by ind_a and ind_b.
The elements of ind_a and ind_b are truncated to integers (with mfloor) and
are used to pick out rows and columns, respectively. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind_a = [3.14 1.99] and ind_b = [2.1],
then the result would be a matrix of the form [8; 2] (that is, elements (3,2)
and (1,2)). This is analoguous to the MATLAB statement, mat(ind_a, ind_b).
arguments:
matrix *mat   matrix to be indexed
matrix *ind_a   row indexing matrix
matrix *ind_b   column indexing matrix
returns:
matrix * the indexed input matrix, NULL if error.
usage:
B = index_rows_cols(mat, ind_a, ind_b); // B = mat(ind_a, ind_b)
MATLAB equivalent:
>> B = mat(ind_a, ind_b)
```

```
see also:
index_rows(), index_cols(), sub_matrix()
note:
For greatest convenience, use sub_matrix() for all indexing purposes.
*/
matrix *index_rows_cols(matrix *mat, matrix *ind_a, matrix *ind_b);
/*  matrix *sub_matrix(matrix *mat, matrix *rows, matrix *cols);
description:
Returns the rows and columns of mat that are pointed out by (the matrices)
rows and cols.
The elements of rows and cols are truncated to integers (with mfloor) and
are used to pick out rows and columns, respectively. That is,
suppose mat = [1 2 3; 4 5 6; 7 8 9], and rows = [3.14 1.99] and cols = [2.1],
then the result would be a matrix of the form [8; 2] (that is, elements (3,2)
and (1,2)). This is analogous to the MATLAB statement, mat(rows, cols).
To just index rows, like MATLAB's mat(rows, :), set cols to NULL. Similary,
to just index columns, like MATLAB's mat(:, cols), set rows to NULL. By using
this scheme all indexing can be done with sub_matrix().
arguments:
matrix *mat   matrix to be indexed
matrix *rows  row indexing matrix
matrix *cols  column indexing matrix
returns:
matrix * the indexed input matrix, NULL if error.
usage:
B = sub_matrix(mat, rows, cols); // B = mat(rows, cols)
B = sub_matrix(mat, rows, NULL); // B = mat(rows, :)
B = sub_matrix(mat, NULL, cols); // B = mat(:, cols)
MATLAB equivalent:
>> B = mat(rows, cols)
see also:
index_rows(), index_cols(), index_rows_cols()
note:
For greatest convenience, use sub_matrix() for all indexing purposes.
*/
matrix *sub_matrix(matrix *mat, matrix *rows, matrix *cols);
/*  matrix *assign(matrix *target, matrix *rows, matrix *cols, matrix *source)
description:
Puts the matrix source into a submatrix of target indicated by rows and cols.
That is, rows and cols defines a submatrix of target (exactly like sub_matrix())
and this submatrix is overwritten with data from the source matrix. This is
analogous to the MATLAB statement target(rows, cols) = source. Needless to
say, the submatrix of target and source must be of the same dimensions.
For example, suppose
target = [1  2  3  4;
   5  6  7  8;
   9 10 11 12],
source = [13 14;
          15 16],
    rows = [3 2] and cols = [1 2], the resulting matrix would be
[ 1  2  3  4;
 15 16  7  8;
 13 14 11 12 ].
If rows or cols is NULL, this means all the rows and all the columns of target.
That is, target(rows,:) = source would be coded as
assign(target, rows, NULL, source), and similary, target(rows,:) = source
would be coded as assign(target, rows, NULL, source).
arguments:
matrix *target matrix to be written to
matrix *rows row indexing matrix
matrix *cols column indexing matrix
matrix *source matrix whose data will be written to target.
returns:
matrix * copy of target, NULL if error.
usage:
```

```
assign(target, rows, cols, source) // target(rows, cols) = source
assign(target, rows, NULL, source) // target(:, cols) = source
assign(target, NULL, cols, source) // target(rows, :) = source
see also:
sub_matrix() temp_copy()
note:
Does not handle the case target(:,:) = source. For this use
target = temp_copy(source).
*/
matrix *assign(matrix *target, matrix *rows, matrix *cols, matrix *source);

/*  matrix *range(FLOATTYPE from, FLOATTYPE step, FLOATTYPE to);
description:
Creates a vector like MATLAB's from:step:to. If step is 0, then it is set to 1.
For example, range(1, 2, 7) results in [1 3 5 7], range(3, 5) results in [3 4 5].
arguments:
FLOATTYPE from start value
FLOATTYPE step step size
FLOATTYPE to stop value
returns:
matrix * vector with elements starting at from and stopping at to,
spaced by step. NULL if error.
usage:
range(from, step, to) // from:step:to
range(from, 0, to) // from:to
*/
matrix *range(FLOATTYPE from, FLOATTYPE step, FLOATTYPE to);

/*  matrix *scl2mat(double re, double im, int type);
description:
Creates a 1x1 matrix from the scalar (re + j*im), if type is CMPLX. If type is
REAL the imaginary part is ignored.
arguments:
FLOATTYPE re real part
FLOATTYPE im imaginary part
returns:
matrix * 1x1 matrix with the element (re + j*im). NULL if error.
usage:
scalar_mat = scl2mat(3.14, 2.5, CMPLX); //  scalar_mat(1,1) = 3.14 + j*2.5
scalar_mat = scl2mat(3.14, 2.5, REAL);  //  scalar_mat(1,1) = 3.14
*/
matrix *scl2mat(double re, double im, int type);

#endif
```

H.9 MATRIX.C

```
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "types.h"
#include "utils.h"
#include "matrix.h"
#include "list.h"
#include "conv.h"
/* Debugging macro */
#define OOPS printf("oops: %d\n", __LINE__);

/* Private declarations */
static LIST *temp_list;            /* list for temporary matrices */
/* stuff used in op_check and mul_check */
#define COMP 0x4                   /* marks compatible dimesions */
#define SCAL 0x8                   /* marks one operand as a scalar */
#define INT(a) ((int)(a))
```

```
/* Promotes, if necessary, the operands a and b to complex matrices */
int cmplx_promote(matrix *a, matrix *b)
{
int i;
  if (a != NULL && b != NULL)
    {
      if (a->type == CMPLX && b->type == REAL)
        {
          if ((b->im = (FLOATTYPE*)malloc(SIZE(b) * sizeof(FLOATTYPE))) != NULL)
            {
              b->type = CMPLX;
              for (i = 0; i < SIZE(b); i++)
               b->im[i] = 0;
              return 0;
            }
          else
            {
              // error: malloc failure
              error("cmplx_promote: malloc failure");
              return -1;
            }
        }
      if (b->type == CMPLX && a->type == REAL)
        {
          if ((a->im = (FLOATTYPE*)malloc(SIZE(a) * sizeof(FLOATTYPE))) != NULL)
            {
              a->type = CMPLX;
              for (i = 0; i < SIZE(a); i++)
               a->im[i] = 0;
              return 0;
            }
          else
            {
              // error: malloc failure
              error("cmplx_promote: malloc failure");
              return -1;
            }
        }
    }
  return 0;
}
/* Check field type and dimensions. If a or b is a scalar
   then the dimensions are compatible
   op_check is used by add, pmul   */
int op_check(matrix *a, matrix *b)
{
  int type = 0;
  matrix *tmp;
  if (a != NULL && b != NULL)
    {
      if (a->type == CMPLX || b->type == CMPLX)
        /* Need for complex promotion? */
        {
          cmplx_promote(a, b);
          type |= CMPLX;
        }
      else
        type |= REAL;
      if (a->rows == b->rows && a->cols == b->cols)
        type |= COMP;          /* a and b are compatible */
      if (SIZE(b) == 1 || SIZE(a) == 1) /* a or b is scalar */
          type |= SCAL | COMP;  /* a and b are compatible */
    }
  return(type);
```

```
}
/* Check field type and dimensions. If a or b is a scalar
   then the scalar is returned in are and aim.
   mul_check is used by mul.  */
int mul_check(matrix *a, matrix *b)
{
  int type = 0;
  matrix *tmp;
  if (a != NULL && b != NULL)
    {
      if (a->type == CMPLX || b->type == CMPLX)
        /* Need for complex promotion? */
        {
          cmplx_promote(a, b);
          type |= CMPLX;
        }
      else
        type |= REAL:
      if (a->cols == b->rows)
        type |= COMP;                /* a and b are compatible */
      if (SIZE(b) == 1 || SIZE(a) == 1) /* a or b is scalar */
          type |= SCAL | COMP;   /* a and b are compatible */
    }
  return(type);
}
/****************** Memory management of matrices *******************/
/* Allocates memory for a matrix with dimension rows and cols */
matrix *new_matrix(int rows, int cols, int type)
{
  matrix *mat;
  mat = NULL;
  /* Try to allocate matrix */
  if ((mat = (matrix*)malloc(sizeof(matrix))) != NULL)
    {
      mat->cols = cols; mat->rows = rows; mat->type = type;
      /* Try to allocate real part */
      if ((mat->re = (FLOATTYPE*)malloc(SIZE(mat) * sizeof(FLOATTYPE))) != NULL)
        {
          if (type == CMPLX)    /* Try to allocate imaginary part */
            if ((mat->im = (FLOATTYPE*)malloc(SIZE(mat) * sizeof(FLOATTYPE))) == NULL)
              {                      /* malloc of mat->im failed, cleanup */
                free(mat->re);
                free(mat);
                mat = NULL;
              }
        }
      else                          /* malloc of mat->re failed, cleanup */
        {
          // error: malloc failure
error("new_matrix: malloc failure");
          free(mat);
          mat = NULL;
        }
    }
  else                              /* malloc of mat failed, cleanup */
    {
      // error: malloc failure
error("new_matrix: malloc failure");
      return(NULL);
    }
  return(mat);
}
/* Frees memory used by m */
```

```
void kill_matrix(matrix *m)
{
  if (m != NULL)
    {
      if (m->re != NULL)
        free(m->re);
        if (m->type == CMPLX)
        if (m->im != NULL)
          free(m->im);
    }
}
/* Returns a copy the matrix source */
matrix *copy_matrix(matrix *source)
{
  matrix *target;
  FLOATTYPE *tre, *sre;
  int i;
  if ((target = new_matrix(source->rows, source->cols, source->type)) != NULL)
    {
      sre = source->re;
      tre = target->re;
      for (i = 0; i < SIZE(source); i++)
        tre[i] = sre[i];
      if (source->type == CMPLX)
        {
          sre = source->im;
          tre = target->im;
          for (i = 0; i < SIZE(source); i++)
            tre[i] = sre[i];
        }
      return(target);
    }
  return(NULL);
}

/*************** Routines for temporary matrices ***************/
/* Allocates memory for a temp matrix with dimension rows and cols */
matrix *new_temp(int rows, int cols, int type)
{
  matrix *mat;
  if ((mat = new_matrix(rows, cols, type)) != NULL)
    {
      if (appendlist(temp_list, mat) == NULL)
        {
        return(NULL);
        }
    }
  return(mat);
}
/* Copies matrix m to a temp matrix */
matrix *copy_temp(matrix *source)
{
  matrix *target;
  if ((target = copy_matrix(source)) != NULL) /* copy source matrix */
    {
      if (appendlist(temp_list, target) == NULL) /* Try to insert target in temp_list */
        {
        kill_matrix(target);
        return(NULL);
        }
    }
  return target;                    /* return copy */
}
/* Frees all memory allocated by temporary matrices */
```

```
void kill_temp_list(void)
{
  matrix *mat;
  rewindlist(temp_list);
  while (mat = (matrix*)walklist(temp_list))
    {
      kill_matrix(mat);
    }
  freelist(temp_list);          /* kill list and... */
  init_temp_list();             /* make a fresh one */
}
/* Initialize the temp matrix list */
int init_temp_list(void)
{
  temp_list = makelist(sizeof(matrix), 5); /* create and initialize
                                                temp_list */
  if (temp_list != NULL)
    return NOERR;
  else
    return -1;
}
/******************* Matrix algebra *****************************/
/* Takes the the real part of m */
matrix *real(matrix *m)
{
  matrix *real;
  if ((real = copy_temp(m)) != NULL)
    {
      if (real->type == CMPLX)
        {
          real->type = REAL;
          if (real->im != NULL)
            {
              free(real->im);
              real->im = NULL;
            }
        }
    }
  return(real);
}
/* Takes the the imaginary part of m */
matrix *imag(matrix *m)
{
  matrix *imag;
  int i;
  if ((imag = copy_temp(m)) != NULL)
    {
      if (imag->type == CMPLX)
        {                       /* if complex get rid of real part */
          if (imag->re != NULL)
              free(imag->re);
          imag->re = imag->im;  /* make old imag part be new real part */
          imag->im = NULL;      /* clean up */
        }
      else
        {                       /* if not complex, the imag part is zero */
          for (i = 0; i < SIZE(imag); i++)
            imag->re[i] = 0;
        }
      imag->type = REAL;        /* the imag part is real */
    }
  return(imag);
```

```
}
/* Return index to maximal element of m */
int max_index(matrix *m)
{
  int i, mx;
  FLOATTYPE *re, *im;
  mx = 0;
  if (m != NULL)
    {
      re = m->re;
      im = m->im;
      if (m->type = REAL)
        {
          for (i = 1; i < SIZE(m); i++)
            {
              if (re[i] > re[mx])
                mx = i;
            }
        }
      else
        {
          for (i = 1; i < SIZE(m); i++)
            {
              if (cabs(re[i], im[i]) > cabs(re[mx], im[mx]))
                mx = i;
            }
        }
    }
  return(mx);
}
/* Return index to minimal element of m */
int min_index(matrix *m)
{
  int i, mi;
  FLOATTYPE *re, *im;
  mi = 0;
  if (m != NULL)
    {
      re = m->re;
      im = m->im;
      if (m->type == REAL)
        {
          for (i = 1; i < SIZE(m); i++)
            {
              if (re[i] < re[mi])
                mi = i;
            }
        }
      else
        {
          for (i = 1; i < SIZE(m); i++)
            {
              if (cabs(re[i], im[i]) < cabs(re[mi], im[mi]))
                mi = i;
            }
        }
    }
  return(mi);
}
/* negates m */
matrix *minus(matrix *m)
{
  int i, size;
  matrix *neg;
  FLOATTYPE *re, *im;
  if (m != NULL)
```

```
    {
      if ((neg = copy_temp(m)) != NULL)
        {
          re = neg->re;
          im = neg->im;
          for (i=0; i < SIZE(m); i++)
            re[i] = -re[i];
          if (m->type == CMPLX)
            {
              for (i=0; i < SIZE(m); i++)
                im[i] = -im[i];
            }
        }
    }
  return(neg);
}
/* conjugate m */
matrix *conj(matrix *m)
{
  int i;
  matrix *conj;
  FLOATTYPE *im;
  if (m != NULL)
    {
      if ((conj = copy_temp(m)) != NULL)
        {
          im = conj->im;
          if (m->type == CMPLX)
            {
              for (i=0; i < SIZE(m); i++)
                im[i] = -im[i];
            }
        }
    }
  return(conj);
}
/* addition */
matrix *add(matrix *a, matrix *b)
{
  matrix *sum;
  FLOATTYPE *sre, *sim, *are, *aim, *bre, *bim;
  int i, type;
  sum = NULL;
  if ( (type = op_check(a, b))  & COMP)
    {
      /* a and b are compatible */
      if (type & SCAL)            /* either a or b is a scalar */
        {
          if (SIZE(b) == 1)             /* b is a scalar */
            {
              are = b->re; aim = b->im; /* interchange a and b */
              bre = a->re; bim = a->im; /* so scalar is in are/aim    */
              sum = copy_temp(a); /* copy the non-scalar to sum */
            }
          else
            {                       /* a is scalar */
              are = a->re; aim = a->im;
              bre = b->re; bim = b->im;
              sum = copy_temp(b); /* copy the non-scalar to sum */
            }
        }
      else                          /* a and b are non-scalars */
        {
          are = a->re; aim = a->im;
          bre = b->re; bim = b->im;
```

```
                sum = copy_temp(b);     /* copy b to sum */
            }
        if (sum != NULL)            /* if copy was successful */
            {
                sre = sum->re;
                sim = sum->im;
                if (type & SCAL)        /* scalar addition */
                    {
                        for (i = 0; i < SIZE(sum); i++)
                            {
                                sre[i] += are[0]; /* scalar is in are/aim... */
                                if (type & CHPLX) /* and sre/sim is a copy of non-scalar */
                                    sim[i] += aim[0];
                            }
                    }
                else
                    {
                        for (i = 0; i < SIZE(sum); i++)
                            {
                                sre[i] += are[i]; /* sre/sim is a copy of b */
                                if (type & CHPLX)
                                    sim[i] += aim[i];
                            }
                    }
            }
    }
    else
    {
      // error: dimension mismatch
      error("add: dimension mismatch");
      }
    return(sum);
}
/* pointwise multiplication */
matrix *pmul(matrix *a, matrix *b)
{
    matrix *prod;
    FLOATTYPE *pre, *pim, *are, *aim, *bre, *bim;
    int i, type;
    prod = NULL;
    if ((type = op_check(a, b)) & COMP)
        {
            /* a and b are compatible */
            if (type & SCAL)            /* either a or b is a scalar */
                {
                    if (SIZE(b) == 1)               /* b is a scalar */
                        {
                            are = b->re; aim = b->im; /* interchange a and b */
                            bre = a->re; bim = a->im; /* so scalar is in are/aim   */
                            prod = copy_temp(a); /* copy the non-scalar to prod */
                        }
                    else                        /* a is scalar */
                        {
                            are = a->re; aim = a->im;
                            bre = b->re; bim = b->im;
                            prod = copy_temp(b); /* copy the non-scalar to prod */
                        }
                }
            else                            /* a and b are non-scalars */
                {
                    are = a->re; aim = a->im;
                    bre = b->re; bim = b->im;
                    prod = copy_temp(b);    /* copy b to prod */
                }
            if (prod != NULL)           /* copy was successful */
```

```
        {
          pre = prod->re;
          pim = prod->im;
          if (type & SCAL)        /* scalar multiplication.          */
            {                     /* scalar is in are/aim and        */
                                  /* pre/pim is a copy of non-scalar */
              for (i = 0; i < SIZE(prod); i++)
                {
                  if (type & REAL) /* real multiplication */
                    pre[i] *= are[0];
                  else            /* complex multiplication */
                    {
                      pre[i] = are[0] * bre[i] - aim[0] * bim[i];
                      pim[i] = are[0] * bim[i] + aim[0] * bre[i];
                    }
                }
            }
          else                    /* ordinary pointwise multiplication */
            {                     /* pre/pim is a copy of b            */
              for (i = 0; i < SIZE(prod); i++)
                {
                  if (type & REAL) /* real multiplication */
                    pre[i] *= are[i];
                  else
                    {             /* complex multiplication */
                      pre[i] = are[i] * bre[i] - aim[i] * bim[i];
                      pim[i] = are[i] * bim[i] + aim[i] * bre[i];
                    }
                }
            }
        }
    else
      {
        // error: dimension mismatch
        error("pmul: dimension mismatch");
      }
    return(prod);
  }
  /* pointwise division */
  matrix *pdiv(matrix *a, matrix *b)
  {
    matrix *prod;
    FLOATTYPE *pre, *pim, *are, *aim, *bre, *bim, tmp;
    int i, type;
    prod = NULL;
    if (a != NULL && b != NULL)
      {
        cmplx_promote(a, b);      /* make sure that types are compatible */
        if (a->rows == b->rows && a->cols == b->cols)
          {
            /* a and b are compatible, ordinary pointwise division */
            if ((prod = new_temp(a->rows, a->cols, a->type)) != NULL)
              {
                pre = prod->re; pim = prod->im;
                are = a->re; aim = a->im;
                bre = b->re; bim = b->im;
                if (prod->type == REAL)
                  {               /* real division */
                    for (i = 0; i < SIZE(prod); i++)
                      {
                        if (bre[i] == 0)
                          {
```

```
                        // error: division with zero
                         error("pdiv: division with zero");
                          return(NULL); /* division with zero, bail out */
                        }
                       tmp *= tmp;
                       pre[i] = are[i] / bre[i];
                     }
                   }
              else
                 {                        /* complex division */
                  for (i = 0; i < SIZE(prod); i++)
                     {
                      if ((tmp = cabs(bre[i], bim[i])) == 0)
                        {
                        // error: division with zero
                         error("pdiv: division with zero");
                          return(NULL); /* division with zero, bail out */
                        }
                       tmp *= tmp;
                       pre[i] = (are[i] * bre[i] + aim[i] * bim[i])/tmp;
                       pim[i] = (aim[i] * bre[i] -are[i] * bim[i])/tmp;
                     }
                 }
             }
         }
  else  /* a or b have not same dimensions but could still be compatible */
     {
        if (SIZE(a) == 1)
          {
            /* a and b are compatible, scalar division */
            if ((prod = new_temp(b->rows, b->cols, b->type)) != NULL)
              {
               pre = prod->re; pim = prod->im;
               are = a->re; aim = a->im;
               bre = b->re; bim = b->im;
               if (prod->type == REAL)
                  {                        /* real division */
                    for (i = 0; i < SIZE(prod); i++)
                      {
                       if (bre[i] == 0)
                         {
                       // error: division with zero
                        error("pdiv: division with zero");
                           return(NULL); /* division with zero, bail out */
                         }
                       pre[i] = are[0] / bre[i];
                      }
                  }
                else
                  {                        /* complex division */
                    for (i = 0; i < SIZE(prod); i++)
                      {
                       if ((tmp = cabs(bre[i], bim[i])) == 0)
                         {
                       // error: division with zero
                        error("pdiv: division with zero");
                           return(NULL); /* division with zero, bail out */
                         }
                       tmp *= tmp;
                       pre[i] = (are[0] * bre[i] + aim[0] * bim[i])/tmp;
                       pim[i] = (aim[0] * bre[i] - are[0] * bim[i])/tmp;
                      }
                  }
              }
          }
     }
```

```
        if (SIZE(b) == 1)
          {
            /* a and b are compatible, scalar division */
            if ((prod = new_temp(a->rows, a->cols, a->type)) != NULL)
              {
                pre = prod->re; pim = prod->im;
                are = a->re; aim = a->im;
                bre = b->re; bim = b->im;
                if (prod->type == REAL)
                  {                   /* real division */
                    if (bre[0] == 0)
                      {
                      // error: division with zero
                      error("pdiv: division with zero");
                      return(NULL); /* division with zero, bail out */
                      }
                    for (i = 0; i < SIZE(prod); i++)
                        pre[i] = are[i] / bre[0];
                  }
                else
                  {                      /* complex division */
                    if ((tmp = cabs(bre[0], bim[0])) == 0)
                      {
                      // error: division with zero
                      error("pdiv: division with zero");
                        return(NULL); /* division with zero, bail out */
                      }
                    tmp *= tmp;
                    for (i = 0; i < SIZE(prod); i++)
                      {
                        pre[i] = (are[i] * bre[0] + aim[i] * bim[0])/tmp;
                        pim[i] = (aim[i] * bre[0] - are[i] * bim[0])/tmp;
                      }
                  }
              }
          }
      }
  return(prod);
}
/* multiplication */
matrix *mul(matrix *a, matrix *b)
{
  matrix *prod;
  FLOATTYPE *pre, *pim, *are, *aim, *bre, *bim, tmpre, tmpim;
  int i, j, k, type;
  prod = NULL;
  if ((type = mul_check(a, b)) & COMP)
    {
      /* a and b are compatible */
      if (type & SCAL)           /* either a or b is a scalar */
        {
          if (SIZE(b) == 1)               /* b is a scalar */
            {
              are = b->re; aim = b->im; /* interchange a and b */
              bre = a->re; bim = a->im; /* so scalar is in are/aim   */
              prod = copy_temp(a); /* copy the non-scalar to prod */
            }
          else
            {                      /* a is scalar */
              are = a->re; aim = a->im;
              bre = b->re; bim = b->im;
              prod = copy_temp(b); /* copy the non-scalar to prod */
            }
```

```
        }                          /* a and b are non-scalars */
    else
    {
        are = a->re; aim = a->im;
        bre = b->re; bim = b->im;
        prod = new_temp(a->rows, b->cols, a->type);   /* copy b to prod */
    }
    if (prod != NULL)              /* copy was successful */
    {
        pre = prod->re;
        pim = prod->im;
        if (type & SCAL)           /* scalar multiplication.        */
            {                      /* scalar is in are/aim and      */
                                   /* pre/pim is a copy of non-scalar */
            for (i = 0; i < SIZE(prod); i++)
                if (type & REAL) /* real multiplication */
                    pre[i] = are[0] * bre[i];
                else
                    {              /* complex multiplication */
                    pre[i] = are[0] * bre[i] - aim[0] * bim[i];
                    pim[i] = are[0] * bim[i] + aim[0] * bre[i];
                    }
        }
        else                       /* ordinary matrix multiplication */
            {                      /* pre/pim is a copy of b          */
            for (i = 0; i < prod->rows; i++)
                {                  /* for ith row of prod */
                if (type & REAL)
                    {              /* real multiplication */
                    for (j = 0; j < prod->cols; j++)/* for jth col of prod */
                        {
                        pre[i + j * prod->rows] = 0;
                        for (k = 0; k < a->cols; k++)
                            {                        /* compute prod(i,j) */
                            pre[i + j * prod->rows] +=
                                are[i + k * a->rows] * bre[k + j * b->rows];
                            }
                        }
                    }
                else
                    {              /* complex multiplication */
                    for (j = 0; j < prod->cols; j++)/* for jth col of prod */
                        {
                        pre[i + j * prod->rows] = 0;
                        pim[i + j * prod->rows] = 0;
                        for (k = 0; k < a->cols; k++)
                            {                        /* compute prod(i,j) */
                            pre[i + j * prod->rows] +=
                                are[i + k * a->rows] * bre[k + j * b->rows]
                                - aim[i + k * a->rows] * bim[k + j * b->rows];
                            pim[i + j * prod->rows] +=
                                are[i + k * a->rows] * bim[k + j * b->rows]
                                + aim[i + k * a->rows] * bre[k + j * b->rows];
                            }
                        }
                    }
                }
            }
        }
else
{
// error: dimension mismatch
```

```
      error("mul: dimension mismatch");
    }
    return(prod);
}
/* append b's rows to a */
matrix *appendrows(matrix *a, matrix *b)
{
    int   ai, bi, ci, i, j;
    FLOATTYPE *cre, *cim, *are, *aim, *bre, *bim;
    matrix *c;
    if (b != NULL)
      {
        if (a == NULL)
          {
            return(b);
          }
        if (a->cols == b->cols)
          {
            cmplx_promote(a, b);
            if ((c = new_temp(a->rows + b->rows, a->cols, a->type)) != NULL)
              {
                are = a->re; aim = a->im;
                bre = b->re; bim = b->im;
                cre = c->re; cim = c->im;
                ai = bi = ci = 0;
                for (j = 0; j < a->cols; j++)
                  {
                    for (i = 0; i < a->rows; i++)
                      cre[ci++] = are[ai++];
                    for (i = 0; i < b->rows; i++)
                      cre[ci++] = bre[bi++];
                  }
                ai = bi = ci = 0;
                if (a->type == CMPLX)
                  for (j = 0; j < a->cols; j++)
                    {
                      for (i = 0; i < a->rows; i++)
                        cim[ci++] = aim[ai++];
                      for (i = 0; i < b->rows; i++)
                        cim[ci++] = bim[bi++];
                    }
              }
            return(c);
          }
        else
    {
// error: dimension mismatch
error("appendrows: dimension mismatch");
}
      }
    return(NULL);
}
/* append b's cols to a */
matrix *appendcols(matrix *a, matrix *b)
{
    FLOATTYPE  *are, *aim, *bre, *bim, *cre, *cim;
    int i, j;
    matrix *c;
    if (b != NULL)
      {
        if (a == NULL)
          {
            return(b);
          }
        if (a->rows == b->rows)
```

```
        {
         if ((c = new_temp(a->rows, a->cols + b->cols, a->type)) != NULL)
            {
             cmplx_promote(a, b);
             are = a->re; aim = a->im;
             bre = b->re; bim = b->im;
             cre = c->re; cim = c->im;
             for (i = 0; i < SIZE(a); i++)
               cre[i] = are[i];
             for (j = 0; j < SIZE(a); j++)
               cre[i + j] = bre[j];
             if (a->type == CMPLX)
                {
                 for (i = 0; i < SIZE(a); i++)
                   cim[i] = aim[i];
                 for (j = 0; j < SIZE(a); j++)
                   cim[i + j] = bim[j];
                }
            }
          return(c);
        }
else
{
// error: dimension mismatch
error("appendcols: dimension mismatch");
}
   }
  return(NULL);
}
/* transpose */
matrix *transp(matrix *m)
{
  matrix *c;
  int i, j;
  c = NULL;
  if (m != NULL)
    {
     if ((c = copy_temp(m)) != NULL)
        {
         c->rows = m->cols;
         c->cols = m->rows;
         for (j = 0; j < m->cols; j++)
           for (i = 0; i < m->rows; i++)
             c->re[j + i * c->rows] = m->re[i + j * m->rows];
         if (m->type == CMPLX)
            {
             for (j = 0; j < m->cols; j++)
               for (i = 0; i < m->rows; i++)
                 c->im[j + i * c->rows] = m->im[i + j * m->rows];
            }
        }
    }
  return(c);
}
matrix *herm(matrix *m)
{
  matrix *c;
  int i, j;
  c = NULL;
  if (m != NULL)
    {
     if ((c = copy_temp(m)) != NULL)
        {
         c->rows = m->cols;
```

```
            c->cols = m->rows;
            for (j = 0; j < m->cols; j++)
              for (i = 0; i < m->rows; i++)
                c->re[j + i * c->rows] = m->re[i + j * m->rows];
            if (m->type == CMPLX)
              {
                for (j = 0; j < m->cols; j++)
                  for (i = 0; i < m->rows; i++)
                    c->im[j + i * c->rows] = -m->im[i + j * m->rows];
              }
          }
      }
    return(c);
}
/* floor function on the matrix m */
matrix *mfloor(matrix *m)
{
  int i;
  if (m != NULL)
    {
      for (i = 0; i < SIZE(m); i++)
        {
          m->re[i] = floor(m->re[i]);
          if (m->type == CMPLX)
            m->im[i] = floor(m->im[i]);
        }
    }
  return(m);
}
/* return all cols of m and rows indexed by a
   Matlab:
   >> m(a,:) % return all cols and rows of m indexed by a
*/
matrix *index_rows(matrix *m, matrix *a) /* All cols, some rows */
      /* matrix *m   source matrix */
      /* matrix *a   index matrix */
{
  matrix *c;
  int i, j, k;
  FLOATTYPE *ci, *ai, *mi;
  if (m != NULL && a != NULL)
    {
      a = mfloor(real(a));                  /* real integer indicies only */
      if (a->re[min_index(a)] > 0
          && a->re[max_index(a)] <= m->rows) /* are indicies in range */
        {
          if ((c = new_temp(SIZE(a), m->cols, m->type)) != NULL)
            {
              k = 0;
              ci = c->re;
              mi = m->re;
              ai = a->re;        /* index real part */
              for (j = 0; j < m->cols; j++) /* for all cols in m */
                for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                  ci[k++] = mi[INT(ai[i]) - 1 + j * m->cols]; /* get column */
              if (m->type == CMPLX) /* index imaginary part if needed */
                {
                  k = 0;
                  ci = c->im;
                  mi = m->im;
                  ai = a->re;
                  for (j = 0; j < m->cols; j++) /* for all cols in m */
                    for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                      ci[k++] = mi[INT(ai[i]) - 1 + j * m->cols]; /* get col */
```

```
            }
        }
        return(c);              /* done */
    }
}
return(NULL);
}
/* return all rows of m and cols indexed by a
   Matlab:
   >> m(:,a) % return all rows and cols of m indexed by a
   >> m(:)   % return m as a column vector (a == NULL)
*/
matrix *index_cols(matrix *m, matrix *a) /* All rows, some cols */
    /* matrix *m;  source matrix */
    /* matrix *a;  index matrix */
{
    matrix *c;
    int i, j, k;
    FLOATTYPE *ci, *mi, *ai;
    if (m != NULL)                  /* if source matrix is not NULL */
    {
        if (a != NULL)              /* if index matrix is not NULL */
        {
            a = mfloor(real(a));            /* real indicies only */
            if (a->re[min_index(a)] > 0
                && a->re[max_index(a)] <= m->rows) /* are indicies in range */
            {
                if ((c = new_temp(m->rows, SIZE(a), m->type)) != NULL)
                {
                    k = 0;
                    ci = c->re;
                    mi = m->re;
                    ai = a->re;    /* index the real part */
                    for (j = 0; j < m->cols; j++) /* for the whole row of m */
                        for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                            ci[k++] = mi[j + (INT(ai[i]) - 1) * m->cols];
                    if (m->type == CHPLX) /* index imaginary part if needed */
                    {
                        k = 0;
                        ci = c->im;
                        mi = m->im;
                        for (j = 0; j < m->cols; j++) /* for whole row of m */
                            for (i = 0; i < SIZE(a); i++) /* for all elem. in a */
                                ci[k++] = mi[j + (INT(ai[i]) - 1) * m->cols];
                    }
                }
                return(c);          /* done */
            }
            else
            {
                // error: indices out of range
                error("index_cols: indices out of range");
            }
        }
        else
        {
            c = copy_temp(m);   /* if a == NULL, make m a row vector, */
            c->rows = SIZE(m); /* this is a matlab convention        */
            c->cols = 1;
            return(c);          /* done */
        }
    }
    return(NULL);
}
/* return m row-indexed by a and column-indexed by b
```

```
    Matlab:
    >> m(a,b) % indexes both rows and cols
    >> m(a)   % indexes m as column vector, but returns a matrix with a's
              % dimensions (b == NULL)
*/
matrix *index_rows_cols(matrix *m, matrix *a, matrix *b)
    /* matrix *m;   source matrix */
    /* matrix *a;   row index matrix */
    /* matrix *b;   column index matrix */
{
  matrix *c;
  int i, j, k;
  FLOATTYPE *ci, *ai, *bi, *mi;
  if (m != NULL && a != NULL)
    {
      a = mfloor(real(a));                /* only real indicies */
      b = mfloor(real(b));
      if (b == NULL)            /* index m as a column vector
                                   and use a's dimensions */
        {
          if (a->re[min_index(a)] > 0
              && a->re[max_index(a)] <= SIZE(m)) /* indicies in range */
            {
              if ((c = copy_temp(a)) != NULL) /* copy a */
                {
                  ci = c->re;
                  mi = m->re;
                  ai = a->re;
                  for (i = 0; i < SIZE(a); i++) /* index m */
                    ci[i] = mi[INT(ai[i]) - 1];
                  if (m->type == CMPLX) /* index imaginary part if needed */
                    {
                      ci = c->im;
                      mi = m->im;
                      for (i = 0; i < SIZE(a); i++) /* index m */
                        ci[i] = mi[INT(ai[i]) - 1];
                    }
                }
              return(c);          /* done */
            }
          else
            {
              // error: indicies out of range
              error("index_rows_cols: indices out of range");
            }
        }
      if (b != NULL)              /* index both rows and cols of m */
        {
          if (a->re[min_index(a)] > 0 && b->re[min_index(b)] > 0 &&
              a->re[max_index(a)] <= m->rows && b->re[max_index(b)] <= m->cols)
            {
              if ((c = new_temp(SIZE(a), SIZE(b), m->type)) != NULL)
                {
                  ci = c->re;
                  mi = m->re;
                  ai = a->re;
                  bi = b->re;
                  k = 0;          /* index real part */
                  for (j = 0; j < SIZE(b); j++) /* for all cols ind. */
                    for (i = 0; i < SIZE(a); i++) /* for all rows ind. */
                      ci[k++] = mi[INT(ai[i])-1 + (INT(bi[j]) - 1) * m->cols];
                  if (m->type == CMPLX) /* index imaginary part if needed */
                    {
                      ci = c->im;
                      mi = m->im;
```

```
                        k = 0;
                        for (j = 0; j < SIZE(b); j++) /* for all cols ind. */
                            for (i = 0; i < SIZE(a); i++) /* for all row ind. */
                                ci[k++] = mi[INT(ai[i]) - 1 +
                                              (INT(bi[j]) - 1) * m->cols];
                    }
                }
                return(c);              /* done */
            }
        else
        {
            // error: indices out of range
            error("index_rows_cols: indices out of range");
        }
    }
}
    return(NULL);
}
/* make matrix of a from:step:to statement,
   Matlab:
   >> from:step:to  %
   >> from:to       % step == NULL -> step size = 1
*/
matrix *range(FLOATTYPE from, FLOATTYPE step, FLOATTYPE to)
{
  matrix *m;
  int i;
  FLOATTYPE j;
  m = NULL;
  if (step == 0)          /* if step == 0, set step size = 1 */
    step = 1;
if (to < from && step < 0)        /* if to < from then step must be < 0 */
{
if ((m = new_temp(1, INT(1 + (from - to) / -step), REAL)) != NULL)
{
j = from;               /* start at from */
for (i = 0; i < SIZE(m); i++) /* for all elem. in m */
        {
            m->re[i] = j;
            j += step;      /* update by step size */
        }
    }
}
else
{                       /* step size > 0 and to > from */
if ((m = new_temp(1, INT(1 + (to - from) / step), REAL)) != NULL)
{
j = from;
        for (i = 0; i < SIZE(m); i++)
          {
            m->re[i] = j;
            j += step;          /* update with step size */
          }
    }
  }
  return(m);                     /* done */
}
/* Returns a copy of mat according to the MATLAB expression
   >>m(row, col)
   For ':' use NULL for row or col, that is
   >>m(row,:)
   would be coded as submatrix(mat, row, NULL) */
matrix *sub_matrix(matrix *mat, matrix *rows, matrix *cols)
{
```

```c
if (mat != NULL)
{
if (rows != NULL && cols != NULL)
return(index_rows_cols(mat, rows, cols)); /* mat(rows, cols) */
if (rows == NULL && cols != NULL) /* mat(:, cols) */
return(index_cols(mat, cols));
if (rows != NULL && cols == NULL) /* mat(rows, :) */
return(index_rows(mat, rows));
return(copy_temp(mat)); /* mat(:,:) */
}
return(mat);
}
/* Assigns source to a submatrix of target indexed by rows and cols.
>>target(rows, cols) = source
rows and cols must be vectors or NULL, NULL is interpreted as :, that is
>>target(rows,:) = source
would be coded as assign(target, rows, NULL, source).
NOTE: does not handle the case
>>target(:,:) = source
use target = copy_temp(source), for this case
*/
matrix *assign(matrix *target, matrix *rows, matrix *cols, matrix *source)
{
FLOATTYPE *r, *c;
int i, j, sr, tr;
if (target != NULL)
{
if (rows != NULL && cols != NULL)  /* target(rows, cols) = source */
{
rows = mfloor(real(rows));  /* Only real integer indicies */
cols = mfloor(real(cols));  /* Only real integer indicies */
                                    /* make sure dimensions match */
if (rows->re[max_index(rows)] <= target->rows     /* max(rows) <= # of rows in target */
    && cols->re[max_index(cols)] <= target->cols /* max(cols) <= # of cols in target */
&& rows->re[min_index(rows)] > 0  /* min(rows) > 0 */
&& cols->re[min_index(cols)] > 0   /* min(cols) > 0 */
&& SIZE(rows) * SIZE(cols) == SIZE(source) )  /* # of elements indexed in target = # of elemer
{
cmplx_promote(target, source);
r = rows->re;
c = cols->re;
sr = source->rows;
tr = target->rows;
for (i = 0; i < SIZE(rows); i++)
for (j = 0; j < SIZE(cols); j++)
{
target->re[INT(r[i]-1) + INT(c[j]-1) * tr] = source->re[i + j * sr];
if (target->type == CMPLX);
target->im[INT(r[i]-1) + INT(c[j]-1) * tr] = source->im[i + j * sr];
}
}
else
{
// error: dimension mismatch
error("assign: dimension mismatch");
}
}
if (rows != NULL && cols == NULL) /* target(rows, :) = source */
{
rows = mfloor(real(rows));  /* Only real integer indicies */
                                    /* make sure dimensions match */
if (rows->re[max_index(rows)] <= target->rows     /* max(rows) <= # of rows in target */
&& rows->re[min_index(rows)] > 0  /* min(rows) > 0 */
&& target->cols == source->cols)                /* target cols = source cols */
```

```
{
cmplx_promote(target, source);
r = rows->re;
sr = source->rows;
tr = target->rows;
for (i = 0; i < SIZE(rows); i++)
for (j = 0; j < target->cols; j++)
{
target->re[INT(r[i]-1) + j * tr] = source->re[i + j * sr];
if (target->type == CMPLX);
target->im[INT(r[i]-1) + j * tr] = source->im[i + j * sr];
}
}
else
{
// error: dimension mismatch
error("assign: dimension mismatch");
}
}
if (rows == NULL && cols != NULL)  /* target(:, cols) = source */
{
cols = mfloor(real(cols));  /* Only real integer indicies */
                                    /* make sure dimensions match */
if (cols->re[max_index(cols)] <= target->cols   /* max(cols) <= # of cols in target */
&& cols->re[min_index(cols)] > 0  /* min(cols) > 0 */
&& target->rows == source->rows)          /* target rows = source rows */
{
cmplx_promote(target, source);
c = cols->re;
sr = source->rows;
tr = target->rows;
for (i = 0; i < target->rows; i++)
for (j = 0; j < SIZE(cols); j++)
{
target->re[i + INT(c[j]-1) * tr] = source->re[i + j * sr];
if (target->type == CMPLX);
target->im[i + INT(c[j]-1) * tr] = source->im[i + j * sr];
}
}
else
{
// error: dimension mismatch
error("assign: dimension mismatch");
}
}
return(target);
}
return(NULL);
}
/* Create a 1x1 matrix from a scalar */
matrix *scl2mat(FLOATTYPE re, FLOATTYPE im, int type)
{
  matrix *tmp;
  if ((tmp = new_matrix(1, 1, type)) != NULL)
    {
      tmp->re[0] = re;
      if (type == CMPLX)
          tmp->im[0] = im;
    }
  return(tmp);
}
```

H.10 UTILS.H

```
#ifndef __utils_h /* Include only once */
#define __utils_h
#include "types.h"

/** Constants for printm() and int_printm() **/
#define PLAIN 0x0  /* Plain format */
#define MATLAB 0x1 /* Print in MATLAB style (with [ ] and ;) */

/* void init_GM(void)
description:
Initializes everything. Clears error string (used by get_error_string()),
sets output print format to MATLAB and 8 digits (see init_print()) and
initializes memory managament (see init_temp_list()).
init_GM() should only be called once and be matched with a close_GM() call.
arguments:
none
returns:
nothing
usage:
init_GM(); // Initialize everything
see also:
close_GM()
*/
void init_GM(void);
/* void close_GM(void)
description:
Frees memory allocated to temporary matrices and cleans up.
close_GM() should only be called once and be matched with a close_GM() call.
Use kill_temp_list() to just free memory allocated by temporary matrices.
arguments:
none
returns:
nothing
usage:
close_GM(); // Clean up
see also:
open_GM()
*/
void close_GM(void);
/* matrix *LV2GM(TD1Hdl re, TD1Hdl im)
description:
Copies Labview matrices to GM matrices. No explicit memory allocation is
necessary, that is handled internally.
arguments:
TD1Hdl re, im Handles to Labview matrix datastructure
returns:
matrix * Gauss machine matrix with data from input arguments re and im
usage:
A = LV2GM(A_real, A_imag);  // Create a GM matrix with real part data from A_real
// and imaginary data from A_imag.
see also:
GM2LV()
*/
matrix *LV2GM(TD1Hdl re, TD1Hdl im);
/* void GM2LV(matrix *A, TD1Hdl re, TD1Hdl im)
description:
Copies data from GM matrix to Labview matrix. Note that re and im must be
already allocated and of correct dimensions. If A is a real matrix then zeros
will be put in im.
arguments:
matrix *a GM matrix whose data is to be copied to re and im
TD1Hdl re, im Handles to Labview matrix datastructure
returns:
nothing
usage:
LV2GM(A, A_real, A_imag);  // Copies data from A into A_real and A_imag.
```

```
see also:
LV2GM()
*/
void GM2LV(matrix *A, TD1Hdl re, TD1Hdl im);

/* init_print(int sdig, int format)
description:
Set number of significant digits and format for printm() and int_printm()
arguments:
int sdig  = number of significant digits
int format  = {MATLAB | PLAIN} output format style
returns:
void
usage:
init_print(6, MATLAB); // 6 significant digits and MATLAB output format
see also:
printm(), int_printm(), init_GM()
*/
void init_print(int sdig, int format);

/* printm(matrix *mat)
description:
prints a matrix to stdout.
arguments:
matrix *mat = matrix to be printed
returns:
void
usage:
printm(Rxx); // prints Rxx to stdout
see also:
init_print() int_printm()
*/
void printm(matrix *mat);

/* int_printm(int_matrix *mat)
description:
prints an integer matrix to stdout.
arguments:
int_matrix *mat = matrix to be printed.
returns:
void
usage:
printm(Rxx); // prints Rxx to stdout
see also:
init_print() printm()
*/
void int_printm(int_matrix *mat);

/* void get_error(char *error_string)
description:
Copies error string to error_string. If an error has occured the error string
will contain an error message. error_string must be allocated to at least 255
characters (by the caller).
arguments:
char *error_string // will contain a copy of the error message
returns:
nothing
usage:
char err_str[255];
get_error(err_str); // get error message
see also:
error(), clear_error(), print_error()
*/
void get_error(char *error_string);

/* void clear_error(void)
description:
Clears the error string. This is typically done at startup or after recovering from
an error.
arguments:
nothing
```

```
returns:
nothing
usage:
clear_error(); // Clear error messages
see also:
get_error(), error(), print_error()
*/
void clear_error(void);
/* void error(char *msg)
description:
Copies the string msg to the global error string. This routine is used to report errors.
The error string can be recovered by get_error(). Maximum length of msg is 255 characters.
arguments:
char *msg; // Error message to be copied to the global error string.
returns:
nothing
usage:
error("Division by zero is a bad idea"); // Post division by zero error message
see also:
get_error(), clear_error(), print_error()
*/
void error(char *msg);
/* void print_error(void)
description:
Prints the global error string to stderr.
arguments:
none
returns:
nothing
usage:
print_error(); // Print error string to stderr.
see also:
get_error(), clear_error(), error()
*/
void print_error(void);
#endif
```

H.11 UTILS.C

```
#include <stdio.h>
#include <string.h>
#include "utils.h"
#include "matrix.h"
static int sigdig = 6;  /* Number of signaificant digits in printouts, default 6 */
static int pformat = PLAIN; /* Format info for print routines */
static char padch = ' ';  /* character to printed between imaginary part and j */
static char err_str[255]; /* string for error messages */

void init_GM(void)
{
/* Initialize everything */
init_temp_list();
init_print(S, MATLAB);
init_conv(21, 6);
clear_error();
}
void close_GM(void)
{
/* Close up everything */
kill_temp_list();
}
/* Convert LV matrices to GM matrices */
```

```
matrix *LV2GM(TD1Hdl re, TD1Hdl im)
{
matrix *tmp;
int rows, cols, i, j;
FLOATTYPE *GMre, *GMim, *LVre, *LVim;
if (re != NULL && im != NULL) /* Check for NULL pointers */
{
rows = (*re)->dimSizes[0];
cols = (*re)->dimSizes[1];
if ( (*im)->dimSizes[0] == 0 )
tmp = new_temp(rows, cols, REAL);
else
tmp = new_temp(rows, cols, CMPLX);
if (tmp != NULL)
{
HLock(re);  /* Be safe */
LVre = (*re)->arg1;   /* Dereference data pointers for real part */
GMre = tmp->re;
for (i = 0; i < rows; i++) /* Copy real LV data to GM data */
for (j = 0; j < cols; j++)
GMre[i + j * rows] = LVre[i * cols + j];
if (tmp->type == CMPLX)
{
HLock(im); /* Be safe */
LVim = (*im)->arg1; /* Dereference data pointers for real part */
GMim = tmp->im;
for (i = 0; i < rows; i++) /* Copy complex LV data to GM data */
for (j = 0; j < cols; j++)
GMim[i + j * rows] = LVim[i * cols + j];
HUnlock(im);
}
HUnlock(re);
}
}
return(tmp);
}
/* Convert GM matrices to LV matrices */
void GM2LV(matrix *a, TD1Hdl re, TD1Hdl im)
{
int rows, cols, i, j;
FLOATTYPE *GMre, *GMim, *LVre, *LVim;
if (a != NULL && re != NULL && im != NULL) /* Check for NULL pointers */
{
rows = (*re)->dimSizes[0];
cols = (*re)->dimSizes[1];
if (a->rows == rows && a->cols == cols) /* make sure dimensions match */
{
HLock(re); HLock(im); /* Be safe */
LVre = (*re)->arg1;
LVim = (*im)->arg1;
GMre = a->re;
GMim = a->im;
for (i = 0; i < rows; i++)
{
for (j = 0; j < cols; j++)
{
LVre[i * cols + j] = GMre[i + j * rows];
if (a->type == CMPLX)
LVim[i * cols + j] = GMim[i + j * rows];
else
LVim[i * cols + j] = 0;
}
}
```

```c
HUnlock(re); HUnlock(im);
}
else
{
// error: dimension mismatch
error("GM2LV: dimension mismatch");
}
}
}

/* Clears error string */
void clear_error(void)
{
strcpy(err_str, "No error reported");
}
/* Copies error string to error_string */
void get_error(char *error_string)
{
strcpy(error_string, err_str);
}
/* Copies msg to err_str */
void error(char *msg)
{
strncpy(err_str, msg, 254);
}
void print_error(void)
{
fprintf(stderr, "%s\n", err_str);
fflush(stderr);
}
void init_print(int sdig, int format)
{
    sigdig = sdig;
pformat = format;
padch = (pformat & MATLAB) ? '*' : ' ';
}
/* Print matrix to stdout */
void printm(matrix *m)
{
  int i, j;
  if (m != NULL)
    {
      if (pformat & MATLAB)
        printf("[");
      for (i = 0; i < m->rows; i++)
        {
          for (j = 0; j < m->cols; j++)
            {
              if (m->type == REAL)
                printf(" % #*.*g", sigdig, sigdig, m->re[i + j * m->rows]);
              else
                printf(" % #*.*g%#+*.*g%cj",sigdig, sigdig, m->re[i + j * m->rows],
                       sigdig, sigdig, m->im[i + j * m->rows], padch);
            }
          if (pformat & MATLAB)
            printf(";");
          printf("\n");
        }
      if (pformat & MATLAB)
        printf("]");
      printf("\n");
    }
}
```

```
/* Print integer matrix to stdout */
void int_printm(int_matrix *m)
{
  int i, j;
  if (m != NULL)
    {
      if (pformat & MATLAB)
        printf("[");
      for (i = 0; i < m->rows; i++)
        {
          for (j = 0; j < m->cols; j++)
            {
              if ((j > 0) && (j % 4 ==0))
                printf("\n");
              if (m->type == REAL)
                printf(" % ld ", m->re[i + j * m->rows]);
              else
                printf(" % ld%+ld%cj ", m->re[i + j * m->rows],
                          m->im[i + j * m->rows], padch);
            }
          if (pformat & MATLAB)
            printf(";");
          printf("\n");
        }
      if (pformat & MATLAB)
        printf("]");
      printf("\n");
    }
}
```

REFERENCES

[1] M. Griffin, F. J. Taylor, and M. Sousa. "New scaling algorithms for the chinese remainder theorem." in *Proc. 22nd Asilomar Conf. on Signals, Syst., and Computers,* 1988.

[2] M. Griffin, M. Sousa, and F. J. Taylor, "Efficient scaling in the residue number system," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing,* 1989.

[3] J. V. Krogmeier and W. K. Jenkins. "Error detection and correction in quadratic residue number systems," in *26th Midwest Symposium on Circuits and Systems,* 1983.

[4] W. K. Jenkins, "The design of error checker for self-checking residue number arithmetic." *IEEE Trans. Computers.* vol. 32, pp. 388–396, Apr. 1983.

[5] G. Zelniker and F. J. Taylor, "A reduced complexity finite field alu," *IEEE Trans. on Circuits and Systems.* vol. 38, pp. 1571–1573, Dec. 1991.

[6] A. van de Goor and C. Verruijt. "An overview of deterministic functional ram chip testing." *ACM Computing Surveys,* vol. 22, pp. 5–34, March 1990.

[7] S. Y. Kung. "VLSI array processors." *IEEE ASSP Magazine,* pp. 4–22, July 1985.

[8] E. Arnould, H. T. Kung, O. Menzilcioglu, and K. Sarocky, "A systolic array computer," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing,* 1985.

[9] H. T. Kung and others:. "iWarp: an integrated solution to high-speed parallel computing." *IEEE Trans. Computers,* vol. 38, pp. 330–339, Sep. 1988.

[10] R. Simar. "The TMS320C40: a DSP for parallel processing," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing,* pp. 1089–1092, 1991.

[11] G. H. Golub and C. F. van Loan. *Matrix Computations.* Baltimore: Johns Hopkins University Press. 2nd ed.. 1989.

DISTRIBUTION LIST
Report No. NAWCADWAR-95005-4.5

No. of Copies

Department of Electrical Engineering 2
219 Grinter Hall
University of Florida
Attn: Dr. Fred J. Taylor
Gainsville, FL 32611

Avionics Department 7
Engineering Division (Code 4.5.5.1)
Naval Air Warfare Center
Aircraft Division Warminster
P.O. Box 5152
Warminster, PA 18974-0591
    5 for code 4.5.5.1; Barry J. Kirsch)
    2 for code 7.2.5.5)

Defense Technical Information Center... 2
Attn: DTIC-FDAB
Cameron Station BG5
Alexandria, VA 22304-6145

Center for Naval Analysis 1
4401 Fort Avenue
P.O. Box 16268
Alexandria, VA 22302-0268

Office of Naval Research... 2
800 N. Quincy St.
Attn: Code ONR-313
Arlington, VA 22217-5660